Query-Rewriting-based Database Access for Description logic system

Sachin Manandhar

Submitted in partial fulfillment of the requirements for the degree
Master of Science in Information and Media Technologies

Supervised by
Prof. Dr. Ralf Möller
Prof. Dr. Friedrich Vogt
Dipl.Inform. Michael Wessel

Software Technology and Systems (STS)
Technical University of Hamburg-Harburg (TUHH)

Hamburg, January 2006

ABSTRACT

Knowledge-base management system (KBMS) based on description logics provides semantic organization of data and powerful reasoning service whereas Database Management System (DBMS) provides the easy access and efficient management. KBMS can be used where large amount of data stored in existing relational databases need to be accessed. We present the architecture and algorithms of a system that converts most of the queries into KBMS into a collection of SQL queries. The system thereby rely on the optimization facilities of existing DBMS to gain efficiency.

In our project, we use RACER as a description logic based KBMS and MYSQL as a Database management system. RACER uses nRQL as its Query language to query semantic organization of data defined in A-Box and T-Box. MYSQL uses standard SQL query language to access data maintained in MYSQL database.

DECLARATION

I declare that:

This work has been prepared by myself,

all literal or content based quotations are clearly pointed out,

and no other sources or aids than the declared ones have been used.

Hamburg, January, 2006

Sachin Manandhar

ACKNOWLEDGEMENT

I would like to take this opportunity to thank my project supervisor **Prof. Dr. Ralf Möller,** *Technical University of Hamburg-Harburg*, for his unique way of inspiring students through clarity of thought, enthusiasm and caring. His technical excellence on the subject, unwavering faith and constant encouragement were very helpful to complete this project successfully.

Michael Wessel, my project advisor, is due a special note of thanks for introducing me to the RACER, nRQL, conversion tips and tricks, and for all his guidance and support throughout the project. This project work was enabled and sustained by his vision and ideas.

Thanks to **Prof. Dr. Friedrich Vogt** for being the co-supervisior of this thesis. Finally, I thank you all who had their contribution to this work.

CONTENTS

Chapter 1 .... **1**

Introduction **1**

1.1 nRQL to SQL conversion **3**

1.2 Outline **4**

Chapter 2 .... **6**

RACER and nRQL **6**

2.1 RACER **6**

2.2 nRQL (The New RACER Query Language) **8**

Chapter 3 .... **24**

Database System **24**

Relational Algebra and SQL **27**

Relational Algebra **27**

SQL **32**

Chapter 4 .... **41**

Background on nRQL to SQL conversion **41**

4.1 nRQL to SQL conversion Approach: **43**

4.1.1 Translating primitive concept to SQL query *43*
4.1.2 Translating primitive role to SQL query *44*
4.1.3 Translating complex concepts to SQL query *44*
4.1.3 translating composite descriptions to SQL query *45*

4.2 Problems with the nRQL to SQL translation **46**

Chapter 5 .... **49**

Implementation **49**

5.1 nRQL to SQL query conversion Architecture **49**

5.2 nRQL Parser **50**

5.3 nRQLgraph to SQL query converter **52**

5.3.1 query optimizer *53*
5.3.2 SQL translator *56*

Chapter 6 .... **60**

Benchmarks **60**

6.1 Tools used **60**

6.2 Validation **60**

6.3 Query result comparison and evaluation **64**

6.4 Evaluation **68**

Chapter 7 69
Conclusion 69
7.1 Future work 69
Bibliography 71
Software tools 73
Appendix 73

LIST OF FIGURES

Fig 1.1 RACER knowledge-base interface to DBMS.. **2**

Fig 2.1 Concept hierarchy for the "family" Tbox **9**

Fig 2.2 Depiction of the" ABox smith-family" **9**

Fig 3.1 An Instance of the Person tabl.. .... **25**

Fig 3.2 σ department=STS(E2).. .... **28**

Fig 3.3 π id,department(E2).. .... **29**

Fig 3.4 π id,department(σ department=STS(E2)).. .... **29**

Fig 3.5 Union example.. .... **30**

Fig 3.6 Intersection example.. .... **31**

Fig 3.7 Set difference example.. .... **31**

Fig 3.8 Cross product example.. .... **32**

Fig 3.9 Answer to (SQL query example 1) **34**

Fig 3.10 JOIN example.. **36**

Fig 3.11 OUTER JOIN (left/right) example.. .... **37**

Fig 3.12 OUTER JOIN (full) example.. .... **37**

Fig:4.1 Example database tables (EMPLOYEE, HAS-TEAM) **41**

Fig 4.2 translation for composite concept description.. .... **45**

Fig 5.1 Architecture of nRQL to SQL query conversion... .... **49**

Fig 5.2 Graph representation of "RETRIEVE" query expression **50**

Fig 5.3 Graph representation of simple query body expression **51**

Fig 5.4 Graph representation of compound query body expression **52**

Fig 5.5 Architecture of nRQL graph to SQL query conversion **52**

Fig 5.6 nRQL graph optimization for concepts **52**

Fig 5.7 nRQL graph for (and (?x SUPERVISOR) (?y EMPLOYEE) **55**
(?x ?y HAS-TEAM)) ....

Fig 5.8 optimized nRQL graph for (and (?x SUPERVISOR) **55**
(?y EMPLOYEE) (?x ?y HAS-TEAM)) ....

Fig: 6.1 data values in "employee" table.. **61**

Fig: 6.2 data values in "has_emp" table.. **62**

Fig: 6.3 data values in "has_team" table.. **62**

Chapter 1

Introduction

Many software applications that need the storage and management of large amounts of data rely on DBMS, while applications like expert systems, natural language interfaces, decision support systems rely on knowledge representation systems, which provide better reasoning about more structured data.

The main difference between knowledge representation system and database system is that the latter is oriented to the efficient management of large amounts of data, while the former seek to give a more structured representation of the universe of discourse in which data are placed. More specifically, in some knowledge representation systems, application domains are described by means of a collection of complex terms - or *concepts* - that are placed into a taxonomy. The capability of *classifying* concepts to form taxonomies, accordingly with well defined semantics, are given by an appropriate calculus. The first goal of classifying concepts is to provide a subsumption algorithm. Concept languages together with semantically grounded subsumption calculi are called description logics. Database systems instead are suited to manage data efficiently with little concern about their dimension, but the formalism for organizing them in a structured way and the capability to infer new information from that already existing information are quite absent.

People have been using DBMS based applications for decades to store and manage their data. DBMS based applications can be widely enhanced by knowledge-base interface. Knowledge-base interface provides deductive extensions to the DBMS based application. The important benefit of knowledge-base interface to DBMS based application is that it enables to create a good intelligent information system.

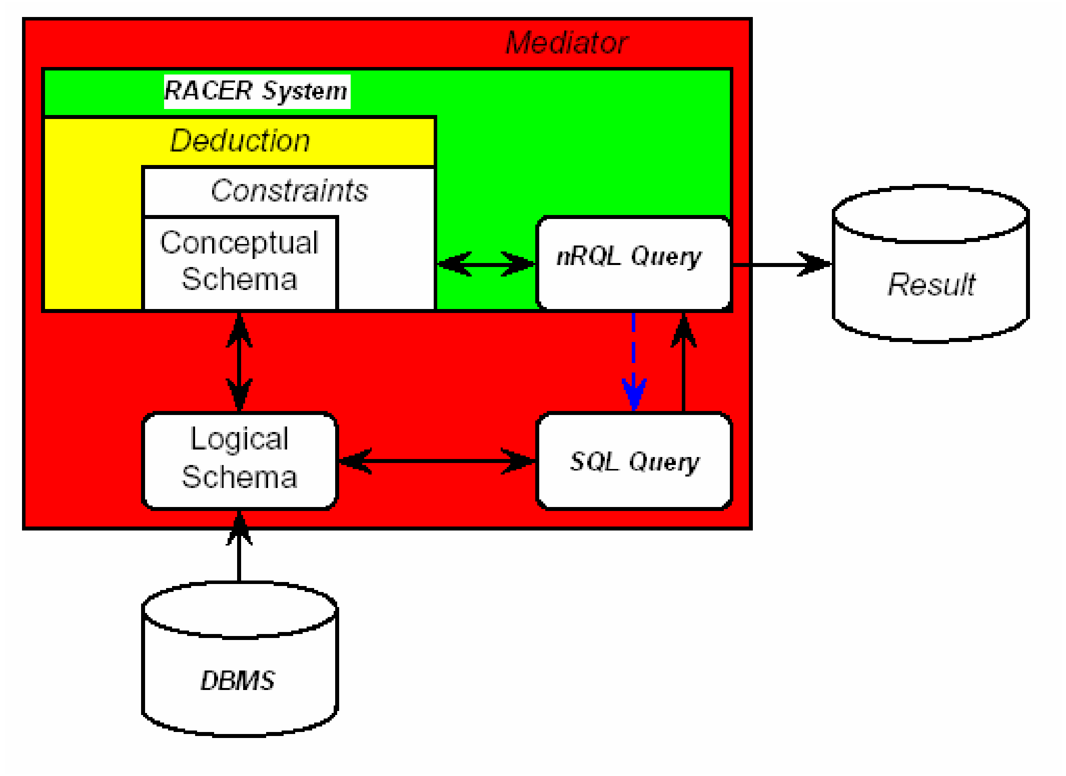


figure 1.1 RACER knowledge-base interface to DBMS

Figure 1.1 shows an implementation of knowledge-base interface. In this figure, RACER knowledge-base management system is used to provide deductive extension to DBMS. Conceptual schema for RACER knowledge-base are built from the logical schema of the database. In RACER new schema can be derived from the conceptual schema.

New schema are called deductive extension of the database schema. This deductive extension of schema enables us to create a good intelligent information system. In this dissertation, we use RACER as knowledge-base management system and MYSQL as DBMS. Knowledge-base interface to DBMS have the following benefits:

<u>1. Large A-Box management in RACER</u>

RACER system lacks efficiency in managing the large amount of data in A-Box. In realistic applications, our experience suggests that knowledge-base system is complex and may involve large number of individuals. It is difficult and sometimes impossible to manage the existing RACER A-Boxes. RACER knowledge-base interface to DBMS enables to store all data about A-Boxes in DBMS, which can be managed better by a DBMS.

2. Knowledge Acquisition from DBMS

The task of acquiring knowledge for a real knowledge-base application often includes a great amount of raw data collection. Knowledge-base interface to DBMS can be used to acquire knowledge from the data which are already stored in the existing databases.

3. Rich concept data modeling

Knowledge-base interface to DBMS can have a richer set of conceptual data model. New and extended data models can be built using the existing data models in knowledge-base system whereas DBMS have fix data-models. Suppose the database have PERSON table and HAS-CHILDREN table. Then the knowledge-base system enables us to define the new concept called PARENT apart from PERSON and HAS-CHILDREN.

In this dissertation, we developed a system for RACER knowledge-base interface to DBMS which converts RACER query language (nRQL) to database query language (SQL). The system enables us to query individuals of deductive extension of database schema and conceptual schema without loading data into A-Box of the RACER. The construction of conceptual schema and its extension in RACER have to be done by the user himself.

1.1 nRQL to SQL conversion

Individuals in RACER are stored in RACER A-Box whereas the information about individuals in DBMS are stored in database tables. It is difficult and sometimes impossible to manage large amount of individuals in RACER A-Boxes. nRQL to SQL conversion system does not load individuals from the RACER A-Box but it loads individuals from the information stored in database tables.

RACER A-Box individuals are more complex as compared to individuals in database tables. RACER A-Box individuals are the instances of concept and roles defined in RACER T-Box . The concept axioms that RACER supports include *concept inclusion* that states the subsumption relationship between two concepts. Therefore individuals of Racer A-Box can be a instance of concept that has subsumption relation between two concept. nRQL query to SQL query converter decompose the complex description of the RACER concepts(or roles) to the level where it can be mapped into SQL query to the table or view in the database. The results from SQL query is returned to the user.

There are certain limitation of this nRQL to SQL conversion tools. It can only handle unary and binary table in database because RACER concepts are unary and roles are binary. In order to handle n-ary table one must decompose the schema or create views in database that have unary or binary relationship.

1.2 Outline

Chapter 2 gives an overview on RACER knowledge management system and RACER query language(nRQL).

Chapter 3 gives an overview on Database Management System, relational algebra and SQL query language.

Chapter 4 discusses previous ideas and proposals to convert subset of nRQL based query language to SQL and formulates the properties that a query language is expected to have.

Chapter 5 gives an overview of implementation and architecture of nRQL query to SQL query converter.

Chapter 6 discusses the benchmarks based on the results of nRQL queries in RACER A-Boxes and the result of converted SQL queries that we have chosen to compare. It discusses compilation of test queries and general methods used to obtain the results.

Chapter 7 summaries the work done and draws a conclusion of this thesis. A number of different ideas are given how this work could be improved, enhanced and effectively used for further practical and research purposes.

Chapter 2

RACER and nRQL

2.1 RACER

RACER (**R**enamed **A**Box and **C**oncept **E**xpression **R**easoner) system was developed by Prof. Dr.Ralf Möller and Volker Haarslev in 1999 at University of Hamburg, Germany. Since then it is being used in many research projects "*as a knowledge representation system that implements a highly optimized tableau calculus for very expressive description logic". [2].*

RACER is a description logic ontology reasoning system supporting DL $\mathcal{ALCQHI}_{R^+}(\mathcal{D}^-)$. RACER extends basic Description Logic $\mathcal{ALC}$ by adding role hierarchies, transitive roles, inverse roles, and qualifying number restrictions. In RACER, the knowledge-base is represented in a tuple (*T-Box*, *A-Box*). *A-Box* contains assertions about individuals and *T-Box* defines the concepts (classes or types of instances), roles (predicates), and features (attributes/properties) of these instances.

RACER system provides reasoning services for multiple *TBoxes* and for multiple *ABoxes*. A collection of concept axioms is called a TBox (**T**erminological **Box**) and a collection of assertional axioms is called an ABox (**A**ssertional **Box**). *[for details ref. A. "Family.RACER" knowledge-base file (TBOX & ABOX), APPENDIX]*

In A-Box, the set of individual names I is the signature of A-Box. The individual set must be disjoint with both the concept set and the role set. In A-Box there are four types of assertions: asserting an individual $\mathcal{IN}$ $\mathcal{IN}_1$ to be of a concept $\mathcal{C}$.; asserting role filler for a role $\mathcal{R}$ to an individual $\mathcal{IN}$ $\mathcal{IN}_2$ that is, individual $\mathcal{IN}$ $\mathcal{IN}_1$ is related to individual $\mathcal{IN}$ $\mathcal{IN}_2$ via role $\mathcal{R}$.); assigning an attribute to an individual; or asserting restriction on an individual. RACER uses optimized tableau algorithm to calculate satisfiability problem.

Tableau algorithm is the dominating algorithm for description logic reasoning currently. The basic idea of this algorithm is to apply transformation rules (these rules preserve the consistency of original A-Box) to A-Box until no rules applicable. If there is no confliction in the A-Box, it is called satisfiable (or consistent). The subsumption problem in RACER is reduced to satisfiability problem.

The RACER T-Box includes the conceptual model of concepts and roles. The model consists of a set of concept names *C* and a set of role names *R*. By exploiting several operators (constructs), one can build complex concepts and roles. RACER supports *Negation, Conjunction,* and *Disjunction* constructs. RACER also provides *Existential* and *Universal* qualified restrictions to ensure that certain roles filler have to be of a specific concept. RACER provides three types of number restrictions: *At-most, At-least,* and *Exactly*. The restrictions can be applied to roles. However, only non-transitive roles (also no transitive sub-roles) can apply cardinality restrictions to attributes.

The concept axioms that RACER supports include *concept inclusion* that states the subsumption relationship between two concepts, *concept equation* that states equivalence between two concepts, and *concept disjointness* that states the disjointness relationship among concepts. RACER can also define the concept name as a special type of a concept term. In RACER, concept axioms can be cyclic or even several axioms for just one concept.

Role declarations in RACER are unique. Only one declaration can be done to one role name. This restriction also applies to attributes in T-Box and individual names in A-Box. Role declarations can declare features (attributes) of a given role, declaring a role to be transitive, and declare hierarchy relationships among roles. In the current version of RACER, the sub-role relationship can not be cyclic.

To suit for variant purposes of reasoning, RACER provides two inference modes. Given a query, RACER can minimize the computation time in the lazy inference mode. If the lazy inference mode is enabled, only the individuals involved in a “direct types” query are realized. However, when the query involves much classifying, another mode – the eager inference mode provides better performance. The other way to save processing time is to save A-Box and T-Box in separate files. Thus, classifying one of them does not need to affect another.

2.2 nRQL (The New RACER Query Language)

nRQL (**n**ew **RACER Q**uery **L**anguage) is the language of RACER for message interchange. It is derived from the previous standard RACER Query Language (RQL). nRQL is RACER expressive ABox query language. nRQL offers support of conjunctive queries. The query variables will be bound to ABox individuals. Queries will make use of concept and role terms. TBoxes supply the vocabulary to be used in the queries.

However, nRQL offers much more than just plan conjunctive queries. The main features of the nRQL language can be summarized as follows:

- Availability of compound (complex) queries built from (simple) query atoms; well defined syntax and clean compositional semantics
- Negation as failure (NAF) semantics as well as true negation available; also roles can be negated with nRQL!
- Support for the concrete domain: availability of complex predicate expressions as well as role chains
- A projection operator for query bodies
- Special support for querying OWL knowledge-bases (e.g., an extended RACER concept syntax for OWL datatype properties, support for the retrieval of told datatype values of OWL datatype properties)
- Complex TBox queries
- Support for so-called hybrid queries[2].

2.2.1 nRQL Language

We introduce the nRQL language with example of the knowledge-base family-1 that comes together with RACER. Figure 2.1 shows the concept hierarchy for the family TBox and figure 2.2 shows the sample description of ABox in smith family.

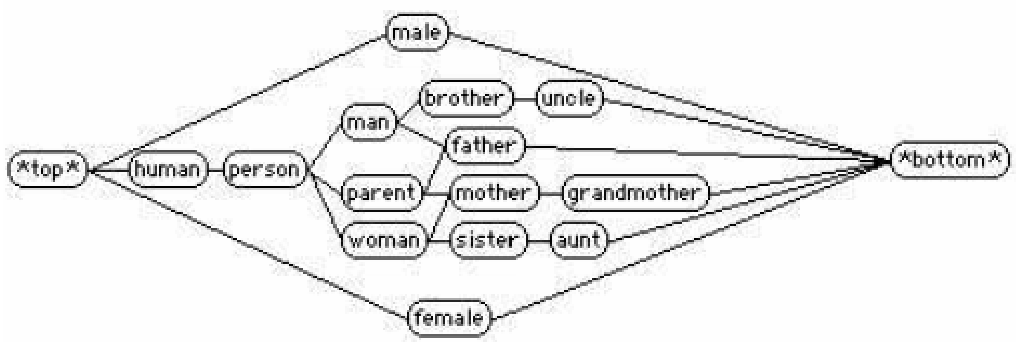


Fig 2.1 Concept hierarchy for the "family" TBox [2]

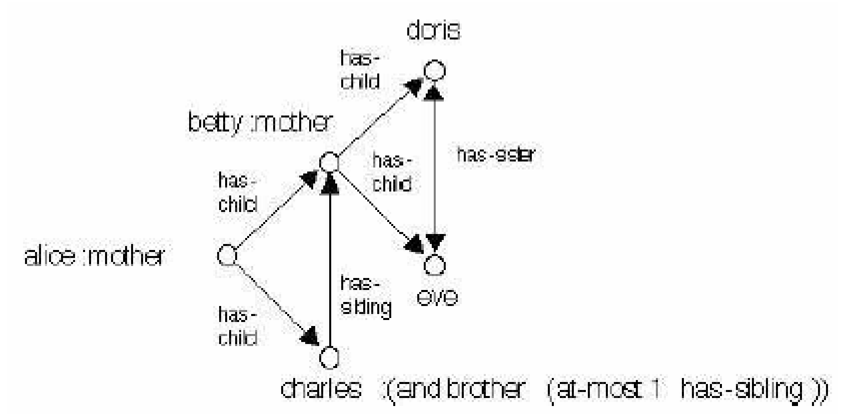


Fig 2.2 Depiction of the" ABox smith-family". [2]

2.2.1.1 Query Atoms

The basic expressions of the nRQL language are called query atoms, or simply atoms. Atoms are either unary or binary. A unary atom references one object, and a binary atom references two objects. An object is either an ABox individual or a variable. Variable begins with a question mark (e.g. ?x) and an individual begins with the instance name (e.g. betty).

There are only three types of atoms available:
1. concept query atoms (which are unary)
2. role query atoms (which are binary)
3. constraint query atoms (which are binary)

Concept Query Atoms

Concept query atom is a unary atom. Suppose we are looking for all instances of woman in the current A-Box of RACER. We simply write:

```
(retrieve  (?x)  (?x  woman))
```

asking RACER for all instances of type woman from the current ABox to be bound to the variable ?x. RACER replies:

```
(((?X  EVE))  ((?X  DORIS))  ((?X  ALICE))  ((?X  BETTY)))
```

i.e., the query is satisfied if the variable ?x is bound to Eve, to Doris, to Alice, or to Betty. RACER has returned a list of binding lists. Each binding list lists a number of variable-value-pairs. Note that there is no guaranty on the order in which the possible bindings are delivered.

Suppose we just want to know if there are any known women at all in the current ABox. We could simply query

```
(retrieve ()  (?x  woman))
```

RACER replies: T, which means
”yes”.

In this case, the list of supplied result objects (the query head) is empty. Such a query never returns any bindings, but only T or NIL (true or false). T is returned if any binding possibility has been found making the query body true; and NIL otherwise.

It is also possible to use ABox individuals within queries. Suppose we want to know if Betty is a woman - we can pose the query

```
(retrieve () (betty woman))
```

If an ABox individual is used which is not present in the ABox, RACER signals an error:

```
(retrieve () (jane woman))
```

yields

```
Error: Undefined individual name JANE in ABox SMITH-FAMILY.
```

[2]

Role Query Atoms

The second type of nRQL atoms are the role query atoms. These are binary atoms, in contrast to the previously discussed unary concept query atoms. It retrieves the instances of the binary pair associated with the role term. Suppose we are looking for all explicitly modeled mother-child-pairs in the current ABox [2]. We can simply write:

```
(retrieve (?mother ?child) (?mother ?child has-child))
```

RACER replies:

```
(((?MOTHER BETTY) (?CHILD DORIS))
 ((?MOTHER BETTY) (?CHILD EVE))
 ((?MOTHER ALICE) (?CHILD BETTY))
 ((?MOTHER ALICE) (?CHILD CHARLES)))
```

The query expression (?mother ?child has-child) is an example of a so-called binary query atom. If we are just interested in the children of Betty (ABox individual) within queries., we could ask RACER like this:

```
(retrieve (?child-of-betty)  (betty  ?child-of-betty  has-child))
```

Role Terms in Role Query Atoms: We mentioned that arbitrary concept expressions can be used in concept query atoms, not only atomic names. The same applies to role query atoms – role terms can be used, not only role names, as the following example demonstrates:

```
(retrieve (?child-of-betty) (?child-of-betty betty (inv has-child)))
```

Again, RACER replies:

```
(((?CHILD-OF-BETTY DORIS)) ((?CHILD-OF-BETTY EVE))).
```

Please note that the set of role terms is rather limited in RACER (only inv is available as a term constructor). However, nRQL adds one more constructor which is only available in role query atoms: negated roles[2].

Negated Roles in nRQL: We know that we can retrieve the instances of the concept (not mother) with (retrieve (?x) (?x (not mother))): RACER replies (((?X CHARLES))), since charles is a man, and man and woman are disjoint. This follows from the definitions of these concepts in the family TBox. Thus, RACER can prove that charles is an instance of the concept (not mother). Usually, in a description logic system, negated roles are not offered, in contrast to negated concepts. Thus, negated roles are neither offered by RACER concept syntax, nor can negated roles be used in ABoxes. However, negated roles can be used for ABox querying with nRQL [2]:

Role query atoms can also be inverted:

```
(retrieve (?mother  ?child)  (inv  (?mother  ?child  has-child)))
```

is equivalent to

```
(retrieve (?mother  ?child)  (?child  ?mother  (inv  has-child)))
```

Of course, the answer will be the same as for

```
(retrieve (?mother  ?child)  (?mother  ?child  has-child)).
```

Constraint Query Atoms

The last type of query atom is the constraint query atom. It is useful in querying concrete domain attribute. It is a binary atom, like the role query atom. These atoms are meant to address the concrete domain part of a KB. Currently, constraint query atoms can only be used on ABoxes whose associated TBoxes do not contain a signature[2]. For example, consider the query

```
(retrieve (?x) (?x (an age)))
```

asking RACER for all instances of those concepts that has an age. RACER replies:

```
(((?X CHARLES)) ((?X DORIS)) ((?X BETTY)) ((?X EVE)) ((?X ALICE)))
```

Similarly, we can also ask RACER who is at least 75 years old:

```
(retrieve (?x) (?x (>= age 75)))
```

RACER replies:

```
(((?X ALICE)))
```

Similarly, we can also find out who is older than whom, or who is older than alice.

The first query can be formulated like this:

```
(retrieve (?x ?y) (?x ?y (constraint age age >)))
```

RACER replies:

```
(((?X CHARLES) (?Y EVE))
((?X CHARLES) (?Y DORIS))
((?X CHARLES) (?Y BETTY))
((?X ALICE) (?Y CHARLES))
((?X ALICE) (?Y EVE))
((?X ALICE) (?Y DORIS))
((?X ALICE) (?Y BETTY))
((?X DORIS) (?Y EVE))
((?X BETTY) (?Y EVE))
((?X BETTY) (?Y DORIS)))
```

That means, Charles is older than Eve, Doris, and Betty, ...

Note that constraint is a keyword, age is a concrete domain attribute, and > is one of the concrete domain predicates offered by RACER.[2]

2.2.1.2 Complex Queries

Complex query expression is combined expression of one or more than one query atoms. Basically, nRQL offers the following constructors which can be used to combine query atoms [2]:

- AND is the standard way of defining compound queries,
- UNION can be used to combine the answers of a set of queries into one answer set, and
- NEG implements a negation as failure semantics; moreover,
- INV can always be used to "reverse" all role query atoms in that subexpression.

AND Queries:

Suppose we want to list all mothers of male persons in the family-1.RACER KB. Whereas the previous queries were all simple (a single unary or binary query atom was sufficient for expressing them), we will now need a compound (or complex) query[2] :

```
(retrieve (?x ?y) (and (?x mother) (?y man) (?x ?y has-child)))
```

RACER replies:

```
(((?X ALICE) (?Y CHARLES)))
```

In this query, we have used the AND operator.

Instead of AND, you can also use CAP or INTERSECTION

Understanding the Query Results It should be noted that

```
(retrieve (?x) (and (?x mother) (?y man) (?x ?y has-child)))
```

is not equivalent to

```
(retrieve (?x) (?x mother)),
```

since the first query (internally) also binds the variable ?y and ensures that (?y man) (?x ?y has-child) holds as well, even if the possible bindings of ?y are not returned to the user.

The objects (variables and individuals) which are referenced within a query body are always bound in every possible way; then the list of result objects is used to determine the format of the output tuples of the query. This can be seen as a projection operation (if we ignore the possibility to duplicate or reorder objects in the output binding lists).

However, the projection to the result objects is always the last step in the query processing chain, and not the first one. Consequently, if the specified list of result objects is empty, we get T iff any binding possibility has been found making the query body true, and NIL otherwise.

Union Queries:

Suppose we want to list all men and woman in the family-1.RACER KB. Whereas the previous queries were all simple (a single unary or binary query atom was sufficient for expressing them), we will now need a compound (or complex) query [2]:

```
(retrieve (?x) (or (?x woman) (?x man)))
```

RACER replies:

```
(((?X CHARLES)) ((?X EVE)) ((?X DORIS)) ((?X BETTY)) ((?X ALICE)))
```

Instead of OR, you can also use CUP or UNION

However, the OR operator is more subtle to understand, since the names of the variables matter. If disjuncts within an OR reference different variables, then the system will ensure that each disjunct references the same variables. For example, the query

```
(retrieve (?x ?y) (or (?x woman) (?y man)))
```

will be internally rewritten (since the first disjunct references ?x, and the second disjunct references?y) into

```
(retrieve (?x ?y) (or (and (?x woman) (?y top))
                      (and (?x top) (?y man)))),
```

ensuring that both disjuncts now bind the same variables (?x ?y), and therefore also have the same arity.

The result will be

```
(((?X EVE) (?Y DORIS))
 ((?X EVE) (?Y CHARLES))
 ((?X EVE) (?Y BETTY))
 ((?X EVE) (?Y ALICE))
 ((?X DORIS) (?Y EVE))
 ((?X DORIS) (?Y CHARLES))
 ((?X DORIS) (?Y BETTY))
 ((?X DORIS) (?Y ALICE))
 ((?X BETTY) (?Y DORIS))
 ((?X BETTY) (?Y EVE))
 ((?X BETTY) (?Y CHARLES))
 ((?X BETTY) (?Y ALICE))
 ((?X ALICE) (?Y DORIS))
 ((?X ALICE) (?Y EVE))
 ((?X ALICE) (?Y CHARLES))
 ((?X  ALICE)  (?Y  BETTY)))
```

As expected, this is the union of the two queries

```
(retrieve  (?x  ?y)  (and  (?x  woman)  (?y  top)))
```

```
(((?X  EVE)  (?Y  BETTY))
 ((?X  DORIS)  (?Y  BETTY))
 ((?X  ALICE)  (?Y  BETTY))
 ((?X  EVE)  (?Y  DORIS))
 ((?X  BETTY)  (?Y  DORIS))
 ((?X  ALICE)  (?Y  DORIS))
 ((?X  EVE)  (?Y  CHARLES))
 ((?X  DORIS)  (?Y  CHARLES))
 ((?X  BETTY)  (?Y  CHARLES))
 ((?X  ALICE)  (?Y  CHARLES))
 ((?X  DORIS)  (?Y  EVE))
 ((?X  BETTY)  (?Y  EVE))
 ((?X  ALICE)  (?Y  EVE))
 ((?X  EVE)  (?Y  ALICE))
 ((?X  DORIS)  (?Y  ALICE))
 ((?X  BETTY)  (?Y  ALICE)))
```

and

```
(retrieve  (?x  ?y)  (and  (?x  top)  (?y  man)))
```

```
(((?X  BETTY)  (?Y  CHARLES))
 ((?X  DORIS)  (?Y  CHARLES))
 ((?X  EVE)  (?Y  CHARLES))
((?X  ALICE)  (?Y  CHARLES)))
```

However, the second disjunct does not produce any new tuples in this example. Consider the query

```
(retrieve  (?y)  (or  (?x  woman)  (?y  man)))
```

Again, it is important to note that this query is *not* equivalent to

```
(retrieve  (?y)  (?y  man)).
```

As already described, RACER will rewrite this query into

```
(retrieve  (?y)  (or  (and  (?x  woman)  (?y  top))
                      (and  (?x  top)  (?y  man)))),
```

Thus, the possible bindings for ?y are from the union of top and man, which is of course top, and not man.

RACER therefore replies:

```
(((?Y  ALICE))  ((?Y  DORIS))  ((?Y  EVE))  ((?Y  CHARLES))  ((?Y  BETTY))),
```

whereas

```
(retrieve  (?y)  (?y  man)).
```

returns

```
(((?Y  CHARLES))),
```

<u>**NEG – The Negation As Failure Constructor**</u>

A NEG constructor is provided which implements a Negation as Failure Semantics. Negation as failure semantics is especially useful for measuring the completeness of the modeling in an ABox, which is important for many applications[2]. Consider the query

```
(retrieve (?x) (?x grandmother))
```

RACER replies:

```
((?X ALICE)).
```

Thus, RACER can prove that Alice is a grandmother. Fine. If we query with a NOT within an ordinary RACER concept term

```
(retrieve (?x) (?x (NOT grandmother))),
```

we get

```
((?X CHARLES)),
```

since Charles is a man, and thus, he can obviously not be grandmother. RACER is able to prove this, given the definitions of man and grandmother in the TBox. However, due to the open world semantics, Charles is the only individual for which RACER is able to prove this. For example, Betty might very well be a grandmother, and we just do not have complete knowledge on Betty. Currently, it is just not known that Betty is a grandmother[2].

So suppose we want to know which individuals are currently not known to be grandmothers. This is were the negation as failure comes into play. Thus, we would like to retrieve all the individuals for which RACER currently cannot prove that they are grandmothers. Consequently, all individuals but Alice (the only known grandmother) should be returned. This is exactly the semantics of a negation as failure atom[2]:

```
(retrieve (?x) (NEG (?x grandmother)))
```

RACER replies:

```
(((?X DORIS)) ((?X EVE)) ((?X CHARLES)) ((?X BETTY)))
```

Note that the NEG is placed “around” the entire atom.

This is simply the complement query of

```
(retrieve (?x) (?x grandmother))
```

w.r.t. the set of all individuals in the ABox:

```
(retrieve (?x) (union (?x C) (neg (?x C))))
```

will always return the set of all ABox individuals, for any concept C.

Note that (?x (not grandmother)) will always return a subset of (neg (?x grandmother)), but not the other way around. This holds for an arbitrary concept term. Things get tricky is negation as failure is used on negated atoms:

```
(retrieve (?x) (neg (?x (not grandmother))))
```

yields

```
(((?X DORIS)) ((?X EVE)) ((?X BETTY)) ((?X ALICE))).
```

Thus, we were asking for all individuals for which RACER cannot prove that they are instances of (not grandmother). Since Charles is the only individual for which RACER can prove this, he is missing from the answer set.[2]

Boolean Complex Queries

In fact, it is possible to combine arbitrarily nested AND, SOME and OR query expressions. We therefore might call the queries boolean. Moreover, the queries are even brought into Disjunctive Normal Form (DNF). Since the DNF might be exponentially larger than the original query and thus result in very big queries, we would like to inform the user of this potential performance pitfall.

In order to understand the query results of some complex queries better it might help if the user reminds him / herself about these internal transformations.[2]

The Projection Operator for Query Bodies

We have already mentioned that the process of constructing the answer tuples for a query can be seen as applying a projection operation. Consider the query

(retrieve (?x) (and (?x c) (?x ?y r) (?y d))).

on the ABox

(instance a c)
(instance b d)

(instance c top)

(related a b r),

which gives us the expected answer

(((?X A))).

In the process of evaluating the query body (and (?x c) (?x ?y r) (?y d)), bindings for ?x and ?y are computed. Internally, nRQL computes an answer set for this query expression, which is simply a list (set) of pairs whose first components give bindings for ?x and whose second components give bindings for ?y. In this example, the internal answer set will be ((A B)). From this set, the final answer (((?X A))) is computed by projecting all the tuples to their first components (the bindings for ?x), since the head of the query is given as (?x). Moreover, reordering, duplication, as well as further specialized head projection operations can be applied to this internal answer set before finally returning the bindings to the user; see also Section 1.1.2 for a discussion of the specialized head projection operators[2].

The nRQL approach of first computing these internal answer sets, and then applying a projection operation to get the desired output according to the form as specified by the query head works fine until the nRQL negation operator (\NOT" or \NEG") is used within the query bodies. To understand the problem, suppose you want to query for the instances of the concept C which do not have a known R successor which is an instance of the concept D. Thus, you want to get the complement of the answer of the previous query: the complement of (((?X A))) is (((?X B)) ((?X C))). Recall that (((?X A))) was the answer to the query (retrieve (?x) (and (?x c) (?x ?y r) (?y d))). In a first attempt to solve this problem you would most likely come up with the following query[2]:

(retrieve (?x) (and (?x c) (?x ?y r) (?y d))),

However, in this query a projection to ?x will be applied to the complement of the set ((A B)), which is the set ((B A) (B C) (C A) (C B) (A C)) { note that the pairs (B B), (A A) and (C C) are missing, due to the UNA for the variables ?x, ?y. Applying the projection to ?x on this set then gives us

(((?X B)) ((?X C)) ((?X A))),

which is not what we wanted. The problem is that the complement operator has been applied to a two-dimensional set, which again yields (naturally) a two-dimensional set, to which then the projection has been applied. In order to get the desired result, a projection operator must be applied before the complement set is constructed, such that a one-dimensional complement is computed, instead of a two-dimensional one.[2]

Chapter 3

Database System

Database is an organized collection of data. It is a collection of records stored in a computer in a systematic way, such that a computer program can consult it to answer questions. For better retrieval and sorting, each record is usually organized as a set of data elements. The items retrieved in answer to queries become information that can be used to make decisions. The computer program used to manage and query a database is known as a database management system (DBMS).

Tables

Tables are the central concept of a database. It is the collection of records, or pieces of knowledge. Typically, for a given table it has a structural description of the type of facts held in that database; this description is known as a schema. The schema describes the objects that are represented in the database, and the relationships among them. An instance of a table is a set of tuples, also called records, in which each tuple has the same number of fields as the table schema.

The schema description for "Person" table can be described as following:

```
CREATE TABLE person (
  id            char(30)
,Name           char(30)
,department     char(30)
,dob            DATE           NOT NULL
);
```

The instance of the "Person" table can be described as following:

Fields (Attributes, Columns)

Touples (records,rows)

ID	Last name	Department	DOB
Sachin	Manandhar	STS	04.11.1979
Michel	Wessel	STS	02.02.1975
Sam	Jones	STS	03.12.1987
Ralph	Muller	STS	06.12.1977

Fig 3.1 An Instance of the Person table

The "Person" table contains six tuples. As we expect from the schema, it has four fields.

Primary Keys

A table requires a key which uniquely identifies each row in the table. This is entity integrity. The key could have one column, or it could use all the columns. No part of a primary key may be NULL. If the rows of the data are not unique, it is necessary to generate an artificial primary key.

The schema description with primary key for "Person" table can be described as following:

```
CREATE TABLE Person (
  id            char(30)
,Name           char(30)
,department     char(30)
,dob            DATE           NOT NULL
,PRIMARY KEY(id)
);
```

Foreign Keys

A foreign key is an attribute that completes a relationship by identifying the parent entity. Foreign keys provide a method for maintaining integrity in the data (called referential integrity) and for navigating between different instances of an entity. Every relationship in the model must be supported by a foreign key. Every dependent and category (subtype) entity in the model must have a foreign key for each relationship in which it participates. Foreign keys are formed in dependent and subtype entities by migrating the entire primary key from the parent or generic entity.

Foreign key attributes are not considered to be owned by the entities to which they migrate, because they are reflections of attributes in the parent entities. Thus, each attribute in an entity is either owned by that entity or belongs to a foreign key in that entity.

The schema description for "has-children" binary table can be described as following:

```
CREATE TABLE has-children (
  parent       CHAR(30)
 ,child        CHAR(30)
,FOREIGN KEY(parent) REFERENCES person (id)
,FOREIGN KEY(child) REFERENCES person (id)

);
```

Relational Algebra and SQL

Relational Algebra

Relational Algebra is the mathematics which underpin SQL operations. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts (one or two) relation instances as arguments and returns a relation instance as a result. This property makes it easy to compose operators to form a complex query – a relational algebra expression is recursively defined to be a relation, a unary algebra operator applied to a single expression, or a binary algebra operator applied to two expressions. We describe the basic operators of the algebra (selection, projection, union, cross-product and difference),as well as some additional operators that can be defined in terms of the basic operators but arise frequently enough to warrant special attention in the following section.

Selection and Projection

Relational algebra includes operator to select row from a relation (σ) and to project columns (π). These operations allow us to manipulate data in a single relation. Consider the instance of the "person" table shown in figurer 3.1 as E2. we can retrieve rows corresponding to STS department employees by using the σ operators.

$$\sigma_{department=STS}(E2)$$

The expression evaluates to the table shown in Figure 3.2. The subscript *department=STS* specifies the selection criterion to be applied while retrieving tuples.

ID	Last name	Department	DOB
Sachin	Manandhar	STS	04.11.1979
Michel	Wessel	STS	02.02.1975
Sam	Jones	STS	03.12.1987
Ralph	Muller	STS	06.12.1977
Tina	Smith	Telematik	06.12.1977
Manish	Ivanov	Telematik	06.12.1977

Fig 3.2 $\sigma_{department=STS}(E2)$

The selection operator σ specifies the tuples to retain through a selection condition. In general, the selection condition is a Boolean combination(i.e., an expression using the logical connectives Λ (AND) and ∨ (OR). Of terms that have the form attribute **op** constant or attribute1 **op** attribute2 , where **op** is one of the comparison operators.

The projection operator π allows us to extract columns from a table; for example, we can find out all person ID, Department by using π. The expression

$$\pi_{id,department}(E2)$$

evaluates to the relation shown in Figure 3.3. The subscript id, department specifies the fields to be retained. The other fields are ‘projected out’.

ID	Department
Sachin	STS
Michel	STS
Sam	STS
Ralph	STS
Tina	Telematik
Manish	Telematik

Fig 3.3 $\pi_{\text{id,department}}(\mathbf{E2})$

Since the result of relational algebra expression is always a relation, we can substitute an expression whenever a relation is expressed. For example we can compute the id and department for person in STS department. The expression

$$\pi_{\text{id,department}}(\sigma_{\text{department=STS}}(E2))$$

produces the result shown in Figure 3.4. It is obtained by applying the selection to E2 and then applying the projection.

ID	Department
Sachin	STS
Michel	STS
Sam	STS
Ralph	STS

Fig 3.4 $\pi_{\textit{id,department}}(\sigma_{\text{department=STS}}(E2))$

Set Operations

The following standard operation on set are also available in relational algebra: union (∪), intersection (∩) , set-difference (-), and cross-product(×).

Union: R ∪ S returns a relation instance containing all tuples that occur either relation instance R or relation instance S (or both). R and S ,must be union – compatible, and the schema of the result id defined to be identical to the schema of R.

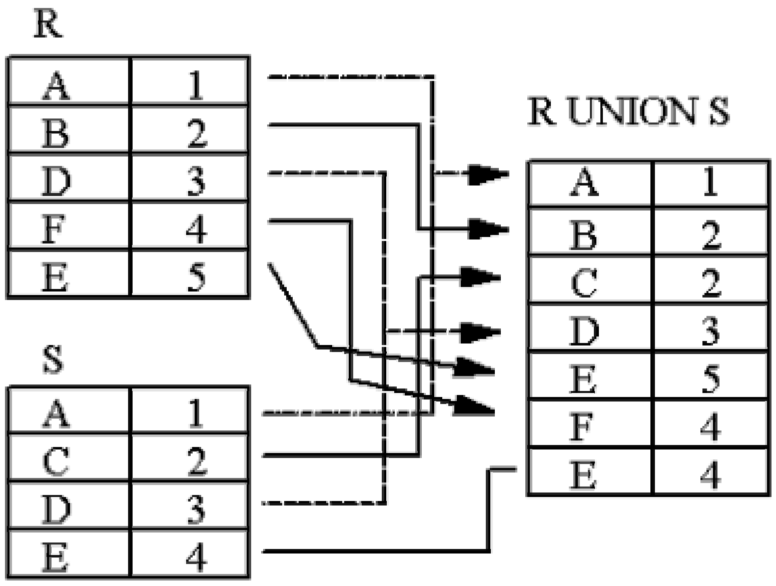
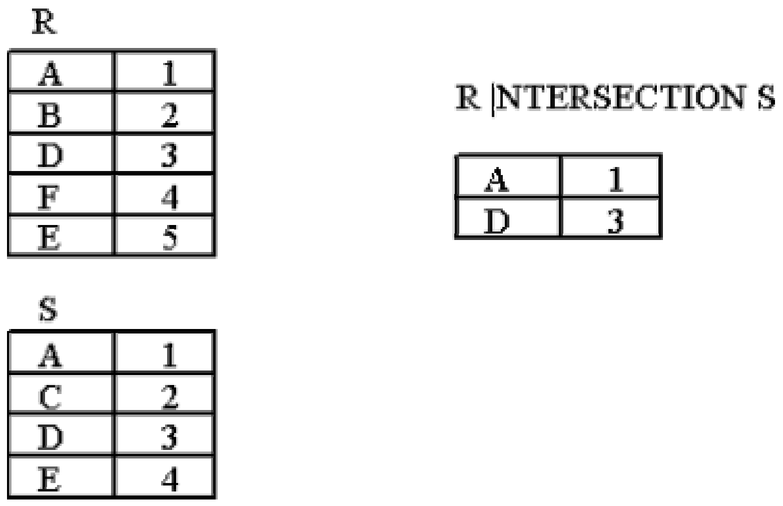


Fig 3.5 Union example

Intersection: R ∩ S returns a relation instance containing all tuples that occur in both R and S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.



R

A	1
B	2
D	3
F	4
E	5

S

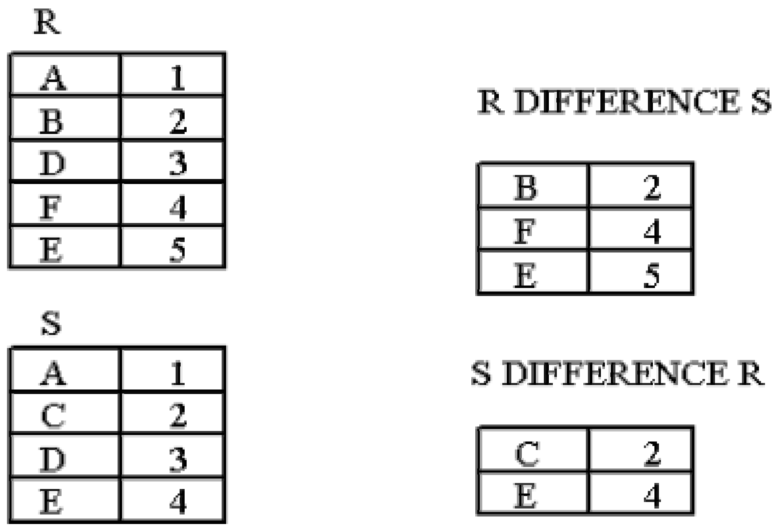
A	1
C	2
D	3
E	4

R |NTERSECTION S

A	1
D	3

Fig 3.6 Intersection example

Set-difference: R – S returns a relation instance containing all tuples that occur in R but not in S. The relations R and s must be union-compatible, and the schema of the result is defined to be identical to the schema of R.



R

A	1
B	2
D	3
F	4
E	5

S

A	1
C	2
D	3
E	4

R DIFFERENCE S

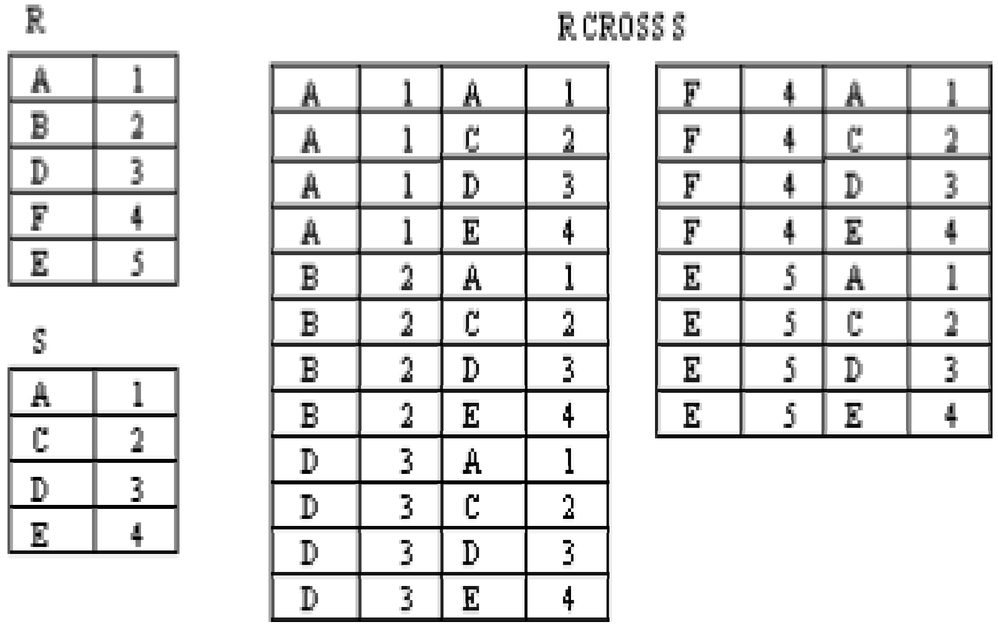
B	2
F	4
E	5

S DIFFERENCE R

C	2
E	4

Fig 3.7 Set difference example

Cross-product: R×S returns a relation instance whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of S (in the same order as they appear in S). The result R × S contains one touple <r,s> (the concatenation of tuples r and s) for each pair of tuples r belongs to R and s belongs to S.



R

A	1
B	2
D	3
F	4
E	5

S

A	1
C	2
D	3
E	4

R CROSS S

A	1	A	1
A	1	C	2
A	1	D	3
A	1	E	4
B	2	A	1
B	2	C	2
B	2	D	3
B	2	E	4
D	3	A	1
D	3	C	2
D	3	D	3
D	3	E	4
F	4	A	1
F	4	C	2
F	4	D	3
F	4	E	4
E	5	A	1
E	5	C	2
E	5	D	3
E	5	E	4

Fig 3.8 Cross product example

SQL

Structured Query Language (SQL) is the most widely used commercial relational database language. It was originally developed at IBM in SEQUEL-XRM and System-R projects (1974-1977). Almost immediately, other vendors introduced DBMS products based on SQL, and it is now a de facto standard. SQL continues to evolve in response to changing needs in the database area. The current ANSI/ISO standard for SQL is called SQL:1999. *[Database Management System, Ramakrishnan, Gehrke, McGrawHill publication 2003]*.

The basic form of SQL query is as follows:

```
SELECT [ DISTINCT ]select-list

FROM from-list

WHERE qualification;
```

Every query must have SELECT clause, which specifies columns to be retained in the result, and a FROM clause, which specifies a cross-product of tables. The optional WHERE clause specifies selection conditions on the tables mentioned in the FROM clause. Such a query intuitively corresponds to a relational algebra expression involving selections, projections and cross-products. Let us consider a simple example.

(SQL query example 1) *Find the id and department of all person.*

```
SELECT DISTINCT E.id, E.department

FROM person E
```

The answer is a set of rows, each of which is a pair <*id, department*>. If two or more person have the same id and department, the answer still contains just one pair. With the id and department. The query is equivalent to applying the projection operator on relational algebra. If we omit the keyword DISTINCT, the answer would be a multi set of rows.

The answer to this query with keyword DISTINCT on instance E of person is shown in Figure 3.9.

ID	**Department**
Sachin	STS
Michel	STS
Sam	STS
Ralph	STS

Tina	Telematik
Manish	Telematik

Fig 3.9 Answer to (SQL query example 1)

Our next query is equivalent to an application of the selection operator of relational algebra.
(SQL query example 2) *Find the id and department of all person.*

```
SELECT DISTINCT E.id, E.department

FROM person as E

Where E.department = ‘STS’
```

This query uses the optional keyword **AS** to introduce a range variable. Incidentally, when we want to retrieve all columns, SQL provides a convenient shorthand: We can simply write SELECT *.

As these two examples illustrate, the SELECT clause is actually used to do projection, whereas selections in the relation algebra sense are expressed using the WHERE clause.

We now consider the syntax of a basic SQL query in more detail.

- The from-list in the FROM clause is a list of table names. A table name can be followed by a range variable; a range variable is particularly useful when the same table name appears more than once in the from-list.

- The select-list is a list of (expressions involving) column names of tables named in the from-list. Column names can be prefixed by a range variable.

- The qualification in the WHERE clause is a Boolean combination (i.e., an expression using the logical connectives AND, OR, and NOT) of conditions of the form expression **op** expression, where **op** is one of the comparison operators {<, <=, =, <>, >=, >}. An expression is a column name, a constant, or an (arithmetic or string) expression.

Although the preceding rule describe (informally) the syntax of a basic SQL query, they do not tell us the meaning of a query. The answer to a query is itself a relation. Which is a multi set of rows in SQL!—whose contents can be understood by considering the following conceptual evaluation strategy:

1. the cross-product of the tables in the from-list.
2. Delete rows in the cross-product that fail the qualification conditions.
3. Delete all columns that do not appear in the select-list.
4. If DISTINCT is specified, eliminate duplicate rows.

This straightforward conceptual evaluation strategy makes explicit the rows that must be present in the answer to the query.

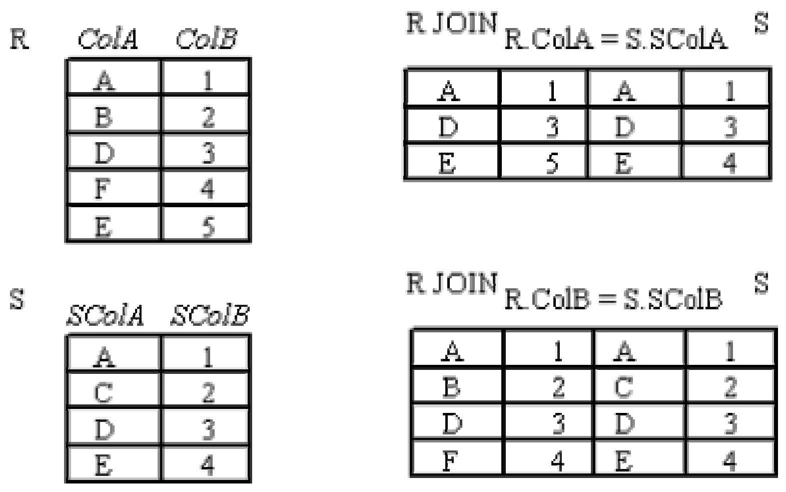
JOIN Operator

JOIN is used to combine related tuples from two or more relations:

- In its simplest form, the JOIN operator is just the cross product of the two or more relations.
- As the join becomes more complex, tuples are removed within the cross product to make the result of the join more meaningful.
- JOIN allows you to evaluate a join condition between the attributes of the relations on which the join is undertaken.

The notation used is

R JOIN$_{join\ condition}$ S



R

ColA	ColB
A	1
B	2
D	3
F	4
E	5

S

SColA	SColB
A	1
C	2
D	3
E	4

R JOIN $_{R.ColA = S.SColA}$ S

A	1	A	1
D	3	D	3
E	5	E	4

R JOIN $_{R.ColB = S.SColB}$ S

A	1	A	1
B	2	C	2
D	3	D	3
F	4	E	4

Fig 3.10 JOIN example

Natural Join

Invariably the JOIN involves an equality test, and thus is often described as an equi-join. Such joins result in two attributes in the resulting relation having exactly the same value. A `natural join' will remove the duplicate attribute(s).

- In most systems a natural join will require that the attributes have the same name to identify the attribute(s) to be used in the join. This may require a renaming mechanism.
- If you do use natural joins make sure that the relations do not have two attributes with the same name by accident.

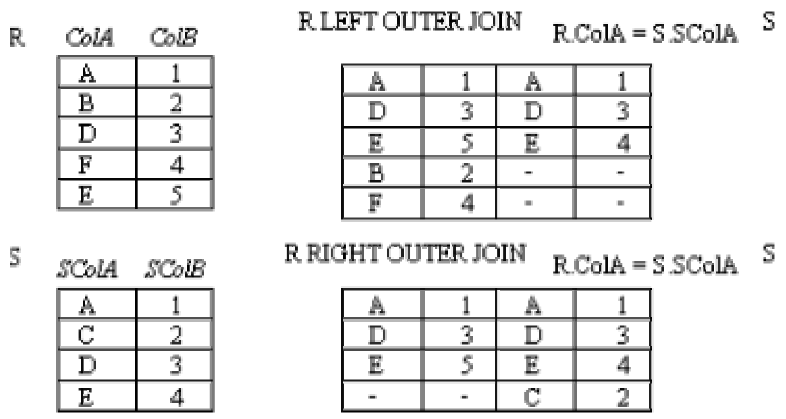
OUTER JOIN

Notice that much of the data is lost when applying a join to two relations. In some cases this lost data might hold useful information. An outer join retains the information that would have been lost from the tables, replacing missing data with nulls.

There are three forms of the outer join, depending on which data is to be kept.

- LEFT OUTER JOIN - keep data from the left-hand table

- RIGHT OUTER JOIN - keep data from the right-hand table
- FULL OUTER JOIN - keep data from both tables



R

ColA	ColB
A	1
B	2
D	3
F	4
E	5

S

SColA	SColB
A	1
C	2
D	3
E	4

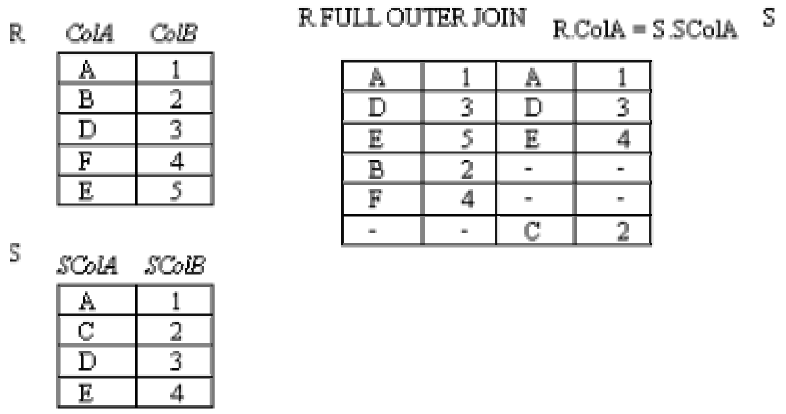
R LEFT OUTER JOIN R.ColA = S.SColA S

A	1	A	1
D	3	D	3
E	5	E	4
B	2	-	-
F	4	-	-

R RIGHT OUTER JOIN R.ColA = S.SColA S

A	1	A	1
D	3	D	3
E	5	E	4
-	-	C	2

Fig 3.11 : OUTER JOIN (left/right) example



R

ColA	ColB
A	1
B	2
D	3
F	4
E	5

S

SColA	SColB
A	1
C	2
D	3
E	4

R FULL OUTER JOIN R.ColA = S.SColA S

A	1	A	1
D	3	D	3
E	5	E	4
B	2	-	-
F	4	-	-
-	-	C	2

Fig 3.12 : OUTER JOIN (full) example

Consider the following SQL to find parent of the STS department.

```
SELECT DISTINCT parent.id
FROM  person as parent, has-children, person as child
WHERE child.department = `STS'
  AND parent.id = has-children.parent
  AND child.id = has-children.child
```

The equivalent relational algebra is

PROJECT$_{id}$ (parent JOIN$_{id = parent}$ (
PROJECT$_{parent}$ (has-children JOIN child=id(
SELECT$_{department = \text{`STS'}}$ child)
))

When this query is submitted to the DBMS, its query optimizer tries to find the most efficient equivalent expression before evaluating it.

Subqueries

One SELECT statement can be used inside another, allowing the result of executing one query to be used in WHERE rules of the other SELECT statement. When one SELECT statement appears within another SELECT statement's WHERE clause it is known as a SUBQUERY.

One limitation of subqueries is that it can only return one attribute. This means that the subquery can only have one attribute in its SELECT line. If you supply more than one attribute the system will report an error.

Subqueries are generally used in situations where one might normally use a self join or a view. Subqueries tend to be much easier to understand. For example, if we want to find Who in the database is older than sachin?

```
SELECT id
FROM person
WHERE dob > (SELECT dob FROM person WHERE id = 'sachin')
```

IN and NOT IN for subqueries

IN and Not IN can be used with something like ('BLUE','BLACK')or a subquery returns a similar construct. For example:

```
SELECT regno FROM car
WHERE colour IN (SELECT colour FROM car WHERE person as owner  =
sachin')
;
SELECT regno FROM car
WHERE colour NOT IN (SELECT colour FROM car WHERE person as owner =
'sachin')
;
```

EXISTS

The EXISTS operator is a simple test, which is TRUE if the subquery returns at least 1 row, and FALSE if it return 0 rows. NOT EXISTS does the opposite. For example:

```
SELECT colour
FROM car a
WHERE exists (
        select colour             -- does not matter what is selected
        from car b                -- As we use CAR twice, call this one
b
        where a.colour = b.colour -- CAR rows with the same colour as a
        and   a.regno != b.regno  -- but a car different to the one in
a
     );
```

ANY and ALL

ANY and ALL allows us to handle subqueries which return multiple rows. The subqueries have only a single column which have ANY in front of a query. The rule provided must be true for at least 1 of the rows returned. For example:

```
SELECT regno FROM car
```

```
WHERE colour = ANY (SELECT colour FROM car WHERE person as owner =
'sachin')
;
```

UNION

UNION merge the results of two queries together to form a single output table. UNION only works if each query in the statement has the same number of columns, and each of the corresponding columns are of the same type. For example:

```
SELECT name,count(*)
FROM   driver JOIN car on (name = owner)
UNION
SELECT name,0
FROM   driver
WHERE  name not in (select owner from car)
```

Chapter 4

Background on nRQL to SQL conversion

The syntax of nRQL can be used to define *concepts* (unary relation symbols), *roles* (binary relation symbols), and *individuals* (constants). Atomic sentences about constants can be expressed in nRQL by saying that an individual is an *instance* of a concept, or a role that holds between two individuals.

We explain nRQL to SQL conversion process by the following example. Suppose we have unary table EMPLOYEE and binary table HAS-TEAM in the database.

EMPLOYEE

id
Michel
Sachin
Ralph

HAS-TEAM

supervisor	employee
Michel	Sachin

Figure:4.1 Example database tables (EMPLOYEE, HAS-TEAM)

Let us assume following knowledge-base is being created in RACER, which maps a concept EMPLOYEE to unary table EMPLOYEE and role HAS-TEAM maps to binary table HAS-TEAM.

```
(in-knowledge-base employee employee-database)

(signature :atomic-concepts (employee)
           :roles ((has-team :transitive t))
           :individuals(michel sachin ralph))

(instance sachin employee)
(instance michel employee)
(instance ralph employee)
```

```
(related michel sachin has-team)
```

RACER allows us to create a new complex individuals from the atomic concepts and roles. Here, we create a new concept called SUPERVISOR from EMPLOYEE concept and HAS-TEAM role.

```
(equivalent supervisor (and employee (some has-team employee)))
```

```
(instance michel supervisior)
```

Similarly, complex concepts and complex roles can be built from the atomic ones using different sets of operators to form *term descriptions.* Typical concept forming operators are **AND, OR, NOT, SOME,** etc. Typical role-forming operators are **INVERSE, FEATURE,TRANSITIVE, RANGE.**

In order to simplify our translation process we divide RACER atomic concept into two types of concepts:

1. Primitive concept
2. Complex concept

Primitive concept

Primitive concept in the RACER has its direct mapping into the database tables. For example, the EMPLOYEE concept in RACER has its direct mapping into a unary table EMPLOYEE.

Complex concepts

In contrast to primitive concept, complex concept does not have its direct mapping to database tables but it has description of tables that have its mapping to database tables. For example, the SUPERVISOR concept in RACER does not has its direct mapping into its database table but its description can be mapped through EMPLOYEE table and HAS-TEAM table.

Similarly, to simplify our translation process for roles we consider roles that has its direct mapping to database table. We call this role as a primitive role. For example we consider HAS-TEAM role as a primitive role.

4.1 nRQL to SQL conversion Approach:

Primitive concept and primitive role have its direct mapping to its corresponding unary table and the binary table respectively. The cleanest way to establish a correspondence between values in these tables and the facts in the RACER is to have each tuple in a "concept table" correspond to an individual in the RACER. Similarly, each tuple in a "role table" correspond to a role relationship.

A concept table should have an attribute key which is necessary to distinguish the corresponding individuals. While a role table has attribute keys of both the domain and range of the binary relation. For simplicity, we assume all keys are single values; initially, we have a sole (key) attribute in a concept table and designate it as **id**; and the two attributes of a role table as the *LEFT concept* **id** and the *RIGHT concept* **id**.

4.1.1 Translating primitive concept to SQL query

Since every primitive concept C to be found in RACER corresponds to its respective unary table with definite id. The algorithm to translate RACER primitive concept definition to SQL is written as follows.

```
function getSQL(C)
  var q : QUERY
  q.select := C.id ;
  q.from := Table C;
  q.statement: = "select distinct " + q.select + "from " + q.from;
  return q.statement;
end function
```

4.1.2 Translating primitive role to SQL query

Since every primitive role R to be found in RACER corresponds to its respective binary table with designate left concept and right concept. The algorithm to translate RACER role definition to SQL is written as follows.

```
function getSQL(R)
  var q : QUERY
  var r: ROLE
  q.select := Left.id, RIGHT.id ;
  q.from := Table R, r.LEFTConcept.basetable as LEFT,
               r. RIGHT Concept.basetable as RIGHT;

  q.where := R.left = LEFT.id + “ And “+
             R.Right = RIGHT.id + “ And “+
             LEFT.id + “ In “+ “(“ + getSQL(LEFT) + “)” + “ AND ”
             RIGHT.id + “ In “+ “(“ + getSQL(RIGHT) + “)” + “ AND ”

  q.statement: = “select distinct ” + q.select + “ from “ + q.from;
                  + “ where ” + q.where

  return  q.statement;
end function
```

4.1.3 Translating complex concepts to SQL query

Complex concepts are built using concepts or roles, which have some direct and indirect mapping into its corresponding database table. In order to translate complex concept we need to simplify it into a composite description of simple concept and roles which can be mapped to database. For example we can simplify the SUPERVISOR concept as follows:

SUPERVISOR :: =
 (AND
EMPLOYEE

(SOME has-team EMPLOYEE))

Therefore to generate an equivalent SQL translation for this query, we need to translate the RACER description instead of the composite concept itself. The algorithm to translate composite concept query to SQL query is the following :

```
function getSQL(C)
  var q : QUERY
  q.select := C.id ;
  q.from := Table getPremitiveConcept(C) as C;
  q.where := C.id in getCompositDesc (getRACERDesc(C))
  q.statement: = “select distinct ” + q.select + “from “ + q.from
                + “ where ” + q.where;
 return q.statement;
end function
```

4.1.3 translating composite descriptions to SQL query

It is relatively easy to derive an SQL query that computes the instances of a composite description based on the denotation of its components. For example, if instances of the concepts C and D are computed by the queries Q_C and Q_D, then **and**(C,D) can be obtained by the query Q_C intersect Q_D.

and(C,D)	*getSQL*(C) Intersect *getSQL*(D)
or(C,D)	*getSQL*(C) Union *getSQL*(D)
some(r,c)	select r.left from s_Table z, r_Table r where r.right=z.id and r.right in *getSQL*(C)

Fig 4.2 translation for composite concept description

The algorithm to translate RACER **and**(C,D) composite description is written as follows.

```
        function getCompositDesc(and(C,D))
```

```
  var q : QUERY
  q.statement getSQL(C) + “ Intersect “ + getSQL(D);
  return q.statement;
end function
```

Similarly the algorithm to translate RACER **or**(C,D) composite description is written as following.

```
function getCompositDesc (or(C,D))
  var q : QUERY
  q.statement getSQL(C) + “ UNION “ + getSQL(D);
  return q.statement;
end function
```

Similarly the algorithm to translate RACER **some**(r,c) composite description is written as following.

```
function getCompositDesc (some(R,C))
 var q : QUERY
 q.select := R.left;
 q.from := Table R, RIGHT;
 q.where := R.right=RIGHT.id and RIGHT.ID in getSQL(C)
 q.statement := “Select “ + q.select + “ From “ + q.from
               + “ where ” + q.where;

 return q.statement;
end function
```

4.2 Problems with the nRQL to SQL translation

The above mentioned translation just supports a part of nRQL query language. The other nRQL queries that are not supported by above mentioned translations are:

Queries with individuals

It is also possible to use ABox individuals in nRQL queries. For example, suppose we just want to know if sachin is a EMPLOYEE:

(retrieve () (sachin EMPLOYEE))

Since primitive concept EMPLOYEE corresponds to EMPLOYEE table in database, which has id field as its key. We redefine the algorithm to translate primitive concept with individual as following:

```
function getSQL(C,instance)
  var q : QUERY
  q.select := C.id ;
  q.from := Table C;
  q.where := C.id = instance;
  q.statement: = "select distinct " + q.select + "from " + q.from
                 " where " + q.where;

  return q.statement;
end function
```

Queries with negated concept

It is also possible to use negated concept in nRQL queries. For example, suppose we just want to find individuals which is not a EMPLOYEE:

(retrieve (?x) (?x (not EMPLOYEE)))

The direct translation to this nRQL query would be to query union of all unary tables in database and negate the getSQL (EMPLOYEE). If we represent the union of all unary tables by ANY-OBJECT view query. Then we can evaluate the nRQL query expression (not EMPLOYEE) by getSQL(ANY-OBJECT) MINUS getSQL(EMPLOYEE). In order to increase the performance of the system, we did not implement this algorithm to translate negated concept for our project.

Role Terms in Role Query Atoms:

It is also possible to use arbitrary concept expression in concept query atoms, not only atomic names. The same applies to role query atoms – role terms can be used, not only role names, as the following example demonstrates:

```
(retrieve (?team-of-michel) (?team-of-michel michel (HAS- TEAM)))
```

Although RACER handles this differently, in our project we treat arbitrary concept expression as a variable, assuming that nRQL uses the unique name assumption (UNA) for the variable.

Chapter 5
Implementation

5.1 nRQL to SQL query conversion Architecture

In the following section we are going to discuss the architecture and implementation details of the *nRQL to SQL conversion* project. The current *nRQL to SQL conversion* system consists of two main subprojects. The first subproject is responsible for parsing nRQL query and is implemented as an nRQL graph in memory. The second subproject is the implementation of nRQL graph to SQL query conversion with a RDBMS on its backend.

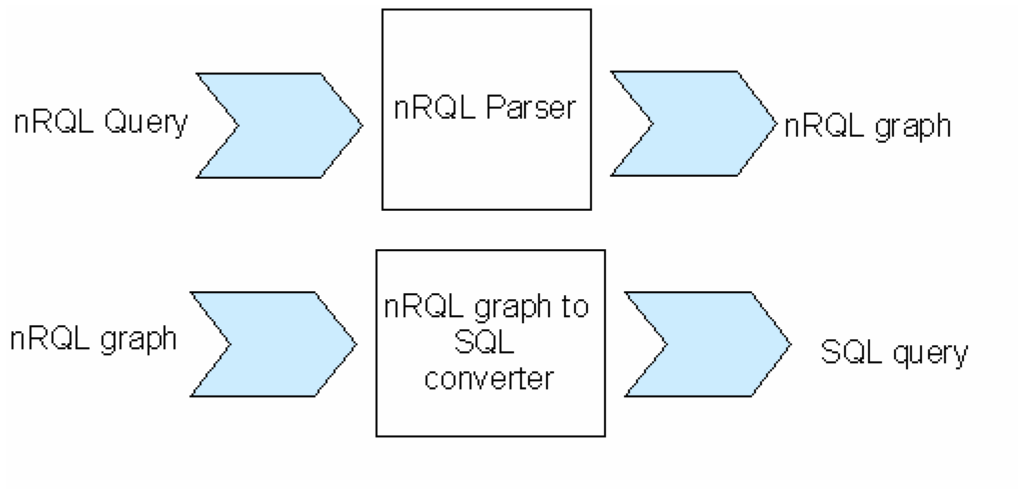


fig 5.1 Architecture of nRQL to SQL query conversion

When nRQL query is passed to query conversion system the parser will evaluate the nRQL query and generate an appropriate nRQL graph in memory for the query. The query converter then maps this graph to SQL query.

5.2 nRQL Parser

nRQL parser splits a text stream, typically a nRQL query, into a graph representation suitable for nRQL to SQL query conversion. It analyzes the stream of tokens and then constructs a graph representation in memory for the nRQL query expression.

The `RETRIEVE` expression is a very important expression of nRQL query. It is composed of Query Head expression and Query Body expression. Query Head expression can be a list of variables or instance of the concepts that is associated with the Query Body expression. Query Body expression can be simple or compound. Simple Query Body expression simply associates query variable or instance to its respective concepts, while compound Query Body expression is composed of one or more Query Body joined together by one or more operators .

The EBNF syntax for `RETRIEVE, Query Head, Query Body expression can be represented as following:`

```
RETRIEVE: <RETRIEVE> “ ” QueryHead()“ ” QueryBody()

Query Head: <OPEN_PAREN> (Var() | “ ” |Qname())* <CLOSE_PAREN>

Query Body: <OPEN_PAREN> (SimpleQueryBody() | CompoundQueryBody())*
            <CLOSE_PAREN>
```

nRQL parser will generate the following graph for this `RETRIEVE operator.`

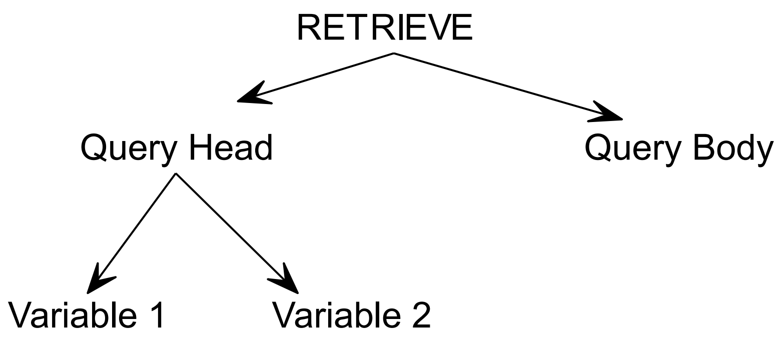


fig 5.2 Graph representation of “RETRIEVE” query expression

5.2.1 Simple Query Body

Simple query body expression is often composed of one or more variables or it could be an instance of the object.

The EBNF syntax for `Simple Query Body can be represented as follows:`

```
Simple Query Body: <OPEN_PAREN> ((var()| “ ” |Qname()))* <CLOSE_PAREN>
```

nRQL parser will generate the following graph for this `Simple Query Body expression.`

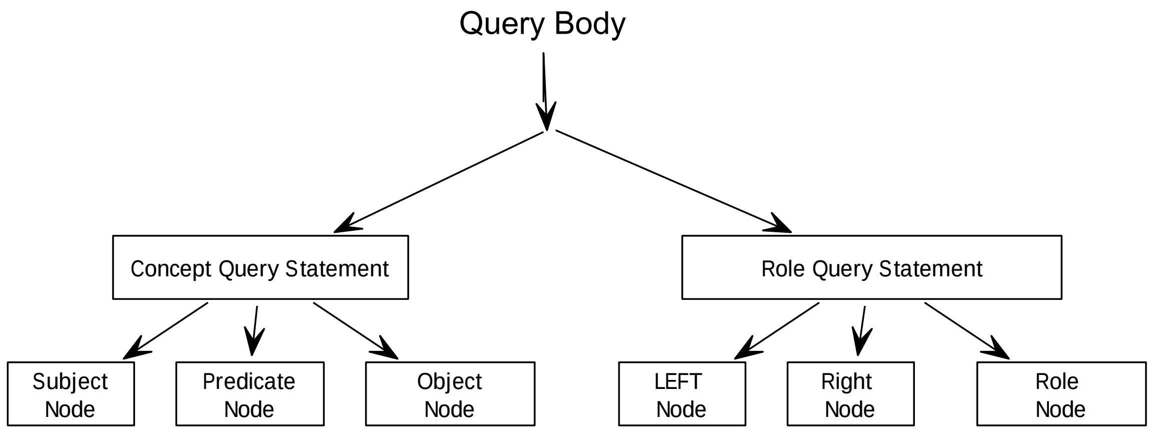


fig 5.3 Graph representation of simple query body expression

5.2.2 Compound Query Body

nRQL compound query statements are often composed of one or more operators. nRQL parser evaluates these operators contained in an nRQL expression to produce an nRQL graph which is an equivalent of the nRQL expression.

The EBNF syntax for `Compound Query Body can be represented as follows:`

```
Compound Query Body: <OPEN_PAREN> (Operator()“ ” QueryBody()
                    “ ” QueryBody ()) <CLOSE_PAREN>
```

nRQL parser will generate the following graph for this `Simple Query Body expression.`

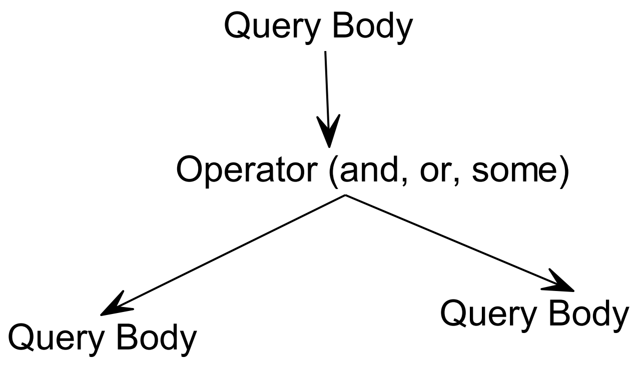


fig 5.4 Graph representation of compound query body expression

5.3 nRQLgraph to SQL query converter

The second sub-project nRQL graph to SQL query converter consists of two components. The first component is called query optimizer. It is responsible for optimizing the nRQL graph produced by the parser, which can be further translated to SQL query. The second component SQL converter analyzes the optimized nRQL graph and produces a SQL output.

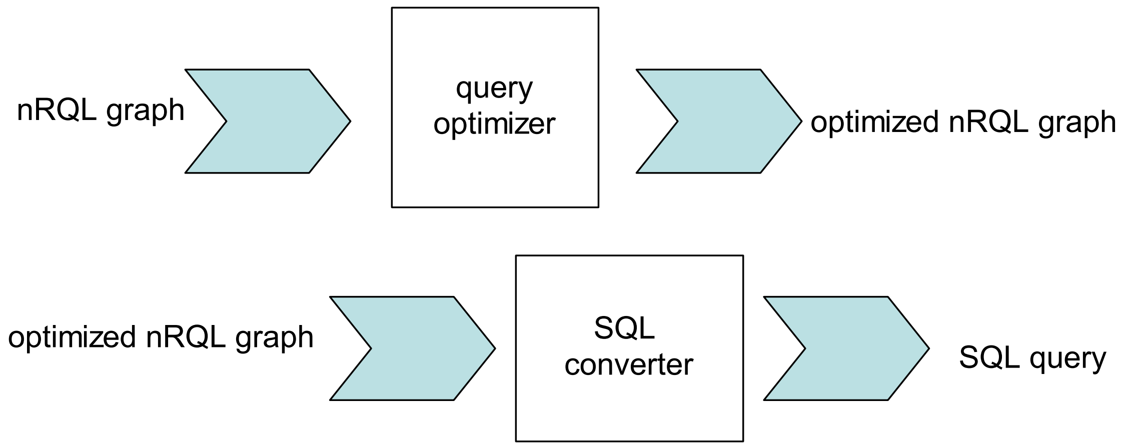


fig 5.5 Architecture of nRQL graph to SQL query conversion

<u>5.3.1 query optimizer</u>

nRQL parser does not distinguish between the primitive concept and compound concept this distinction is done by the query optimizer. Query optimizer traverses every object node and check if the object node is a primitive concept or compound concept against RACER. It uses JRACER to get the concept description from RACER. The following code fragment shows a sample implementation of JRACER.

```
public class Subscription {
     public static void main(String[] argv) {
     RACERServer RACER1 = new
RACERServer("localhost",8088);
     String res;
     try {
          RACER1.openConnection();
          res = RACER1.send("define-concept PARENT");
          RACER1.closeConnection();
          System.out.println(res);
          }
     catch (Exception e) {
               e.printStackTrace();
          }
     }
}
```

If JRACER replies NIL as the concept description for the selected object node, then query optimizer treat this object node as a primitive concept node. The pair of this subject Node, predicate node and the primitive concept node is treated as a triple.

If JRACER replies other than NIL as the concept description query optimizer treat this object node as complex concept. For e.g. (AND EMPLOYEE (SOME HAS-TEAM EMPLOYEE))

Query optimizer refine the graph for the object node. To refine this object node query optimizer connect to nRQL parser and get an equivalent graph for this concept description. The Object node can then be replaced by the graph retrieved from the nRQL parser. The refinement process continues until every object node is represented as a primitive concept node.

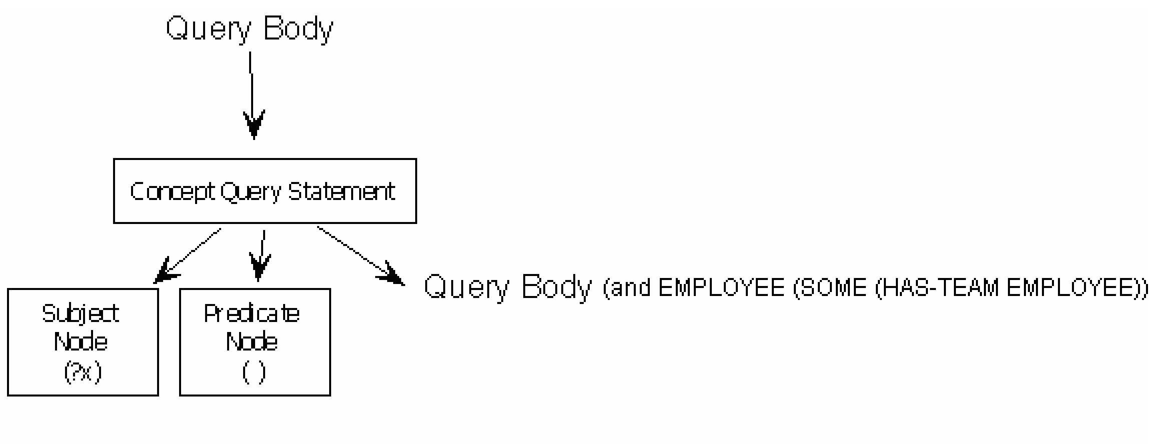


fig 5.6 nRQL graph optimization for concepts

In the case of role based nRQL queries such as

(reterive (?x ?y) (and (?x SUPERVISOR) (?y EMPLOYEE) (?x ?y HAS-TEAM))

nRQL parser generates the graph as following:

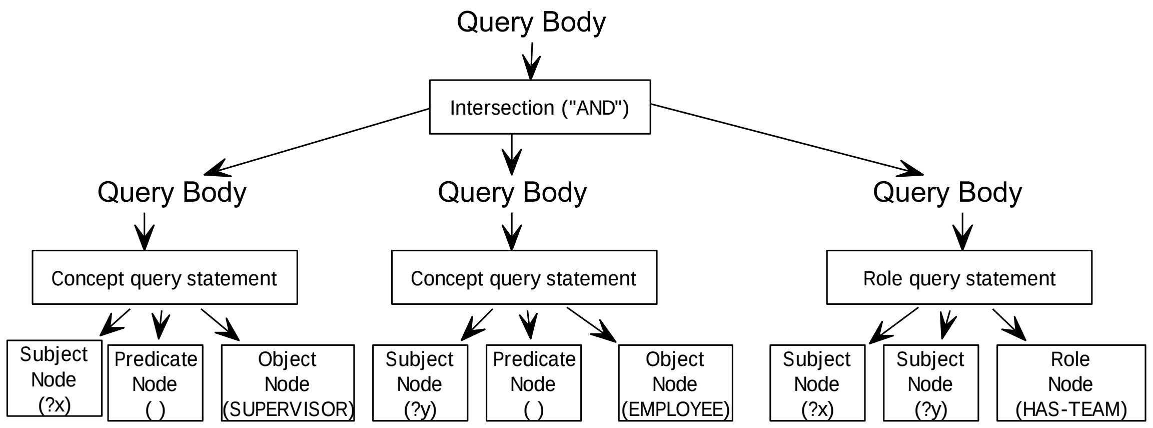


fig 5.7 nRQL graph for (and (?x SUPERVISOR) (?y EMPLOYEE) (?x ?y HAS-TEAM))

Each role query statement is composed of left and right variables. Each of these variable are associated with some concept anywhere in the graph. nRQL graph optimizer searches for the associated concept in the parent nRQL graph and constructs a query body graph as following.

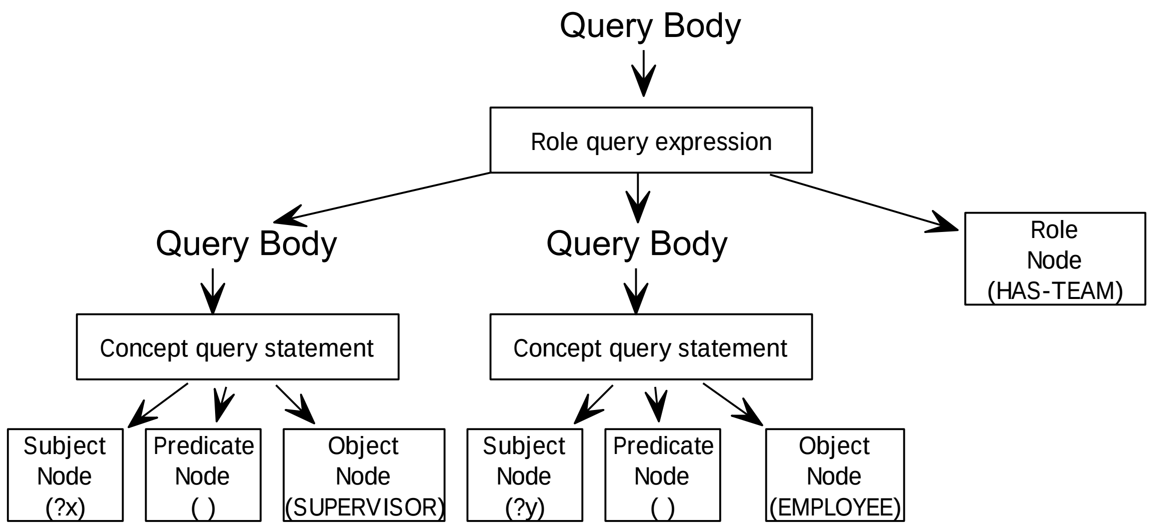


fig 5.8 optimized nRQL graph for (and (?x SUPERVISOR) (?y EMPLOYEE) (?x ?y HAS-TEAM))

Later query optimizer replaces the parent graph of role query body with the optimized graph for role query body. Similar kind of optimization is also done for role based nRQL queries such as (and PERSON (SOME HAS-CHILDREN PERSON)). In this case searching is done in grand-parent graph for concepts and later grand-parent graph is replaced by optimized graph.

5.3.2 SQL translator

Query body of nRQL graph is composed of one or more than one query body sub-graphs. SQL translator define toSQL() function to translate query body nRQL graph toSQL query. toSQL() function always returns a SQL string.

5.3.1.1 SQL translation for Triple

The Level 0 query body of optimized nRQL sub-graph is always a primitive concept query. It has a subject node, a predicate node and a object node with primitive concept. SQL translator define this set as a “Triple” object. The following code snippet of “Triple” object shows an implementation of function toSQL().

```
public String toSQL()
      {
            String SQLString;
            String SubjString;
            String condString;
            SQLString = "Select ";
            condString ="";
            SQLString = SQLString + obj.toString() + ".id";
            SQLString =  SQLString + " from " +
                             obj.toString();

            SubjString =subj.toString();

            char first = SubjString.charAt( 0 );
            if ((first != '?')&& (first != '')){
                  condString = condString +  obj.toString() + ".id = '"
                             + subj.toString() + "'";
                  }

            if (!(condString.equals("")))
                  SQLString = SQLString + " Where " + condString;

            return SQLString;
     }
```

toSQL() function for “Triple” object simply selects object id from object when subject node is an instance of a primitive concept then it selects the object id which is equal to an instance of the primitive concept.

5.3.1.2 SQL translation for role query

Optimized nRQL sub-graph for role query has a left query body, a right query body and a role node. SQL translator define this set as “RoleBaseQuery” object. The following code snippet of as “RoleBaseQuery” object shows an implementation of function toSQL().

```
public String toString(){

      String sqlString;

      if ((!L.isEmpty())&&(!R.isEmpty())){
      String tempLeftObj = L.getTriple().getObject().toString();
      String tempRightObj = R.getTriple().getObject().toString();
      sqlString = "Select ";
      sqlString = sqlString  + tempLeftObj + "," +tempRightObj;
      sqlString = sqlString  + " from ";
      sqlString = sqlString  + roleName;
      String condString="";
      String temp=L.getTriple().getSubject().toString();
      char first = temp.charAt( 0 );
      if ((first != '?')&& (first != '')){
        if (!condString.equals("")){
                  condString = condString + " And ";
                  }
         condString = condString +  tempLeftObj + " = '" + temp + "'";
         }
       temp=R.getTriple().getSubject().toString();
       first = temp.charAt( 0 );
      if ((first != '?')&& (first != '')){
        if (!condString.equals("")){
            condString = condString + " And ";
            }
         condString = condString +  tempRightObj + " = '" + temp + "'";
      }
      if (!condString.equals("")){
                  sqlString = sqlString  + " Where " + condString;
            }
            return sqlString;
                  }
      if ((!L.isEmpty())&&(R.isEmpty())){
            String primaryrole;
            String tempLeftObj = L.getTriple().getObject().toString();
            sqlString = "Select ";
            primaryrole ="";
            DatabaseConnection db = new DatabaseConnection();
            Role role = new Role(db);
            primaryrole = role.getPrimary(roleName,tempLeftObj);
            }
      sqlString = sqlString  + roleName + "."+ primaryrole;
      sqlString = sqlString  + " from ";
      sqlString = sqlString  + roleName + ",";
      sqlString = sqlString  + tempLeftObj;
      sqlString = sqlString  + " Where ";
      sqlString = sqlString  + "(" + roleName + "." + tempLeftObj
                  + " = " + tempLeftObj + ".id" +")";
      sqlString = sqlString  + " And ";
      sqlString = sqlString  + "(";
      sqlString = sqlString +roleName+ "." + tempLeftObj;
      sqlString = sqlString  + " in ";
      sqlString = sqlString  + "(";
      sqlString = sqlString  + L.queryHandler().toSQL();
            sqlString = sqlString  + ")";
            sqlString = sqlString  + ")";
            return sqlString;
            }
 return "";
 }
```

toSQL() function for "RoleBaseQuery" object simply selects role table name, left object table name, and right object table name, and make join query of role table, left table and right table. It also makes sure RoleTable.left is in SQL query of left sub-graph and RoleTable.right is in right sub-graph.

5.3.1.2 SQL translation for compound query

Optimized nRQL sub-graph for compound query always has a left query body and a right query body. SQL translator define this set as "CompoundQuery" object. Compound query could be "And" or "Or" type. The following code snippet of as "CompoundQuery" object shows an implementation of function toSQL().

```
 public String toSQL()
       {
            String SQLString;

           if ((QueryType == 'And')
            sql = sql + L.toSQL()+ " INTERSECT " + R.toSQL();
            }
```

```
        if ((QueryType == 'Or')
        sql = sql + L.toSQL()+ " UNION " + R.toSQL();
        }

    }
```

toSQL() function for “CompoundQuery” object of type “And” simply gives an intersection left sub-graph query and right sub-graph query. Similarly toSQL() function for “CompoundQuery” object of type “Or” simply gives a union of left sub-graph query and right sub-graph query.

Chapter 6

Benchmarks

The aim of this thesis was to design a nRQL to SQL conversion system for a subset of nRQL statement that is powerful enough to queries information in database. This enables us to query instances of the RACER ABox individuals which is stored in DBMS.

6.1 Tools used

We had used various tools to test nRQL queries and SQL queries.

1. RICE (RACER Interactive Client Environment)

RICE is a graphical interface to RACER. It is developed by Vaithi Subramanian. It provides functions for browsing TBoxes and ABoxes, plus querying facilities using the RACER API.

2. SQL Studio and SQLyog

SQL Studio and SQLyog are graphical interface to MYSQL DBMS. It provides functions for creating and browsing database schema, plus multiple querying facilities using the MYSQL MAXDB API.

6.2 Validation

In order to validate our translation. We test the result of SQL query in database against nRQL query in RACER knowledge-base. We consider the following Database tables in MYSQL database.

1. Unary table “employee”
2. Binary table “has_emp”
3. Binary table “has_team”

The schema for unary table “employee” is as follows:

```
CREATE TABLE `employee` (
  `id` char(20) NOT NULL default '',
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

The data value for unary table employee is as follows:

Table employee

id
michel
ralph
sachin
sam

Fig: 6.1 data values in “employee” table

The schema for binary table “has_emp” is as follows:

```
CREATE TABLE `has_emp` (
  `manager` char(20) NOT NULL default '',
  `employee` char(20) NOT NULL default '',
  KEY `manager` (`manager`),
  KEY `employee` (`employee`),
  CONSTRAINT   `has_emp_ibfk_1`   FOREIGN   KEY   (`manager`)
REFERENCES `employee` (`id`),
  CONSTRAINT   `has_emp_ibfk_2`   FOREIGN   KEY   (`employee`)
REFERENCES `employee` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

The data value for binary table “has_emp” is as follows:

Table has_emp

manager	employee
ralph	michel
michel	sachin
michel	sam

Fig: 6.2 data values in “has_emp” table

The schema for binary table has_team is as follows:

```
CREATE TABLE `has_team` (
  `supervisior` char(20) NOT NULL default '',
  `employee` char(20) NOT NULL default '',
  KEY `supervisior` (`supervisior`),
  KEY `employee` (`employee`),
  CONSTRAINT `has_team_ibfk_1` FOREIGN KEY (`supervisior`)
REFERENCES `employee` (`id`),
  CONSTRAINT  `has_team_ibfk_2`  FOREIGN  KEY  (`employee`)
REFERENCES `employee` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

The data value for binary table “has_team” is as follows:

Table has_team

supervisior	employee
michel	sachin
michel	sam

Fig: 6.3 data values in “has_team” table

We then construct a knowledge-base schema in RACER from this database schema. The knowledge-base schema in RACER for above database schema is as follows:

```
 (in-knowledge-base employee employee-database)

(signature :atomic-concepts (employee )
           :roles ((has_emp :transitive t)
               (has_team :transitive t)))
```

We then add individuals to the knowledge-base schema which can be found in employee table. The new knowledgebase with added individuals in RACER A-Box is as follows:

```
(in-knowledge-base employee employee-database)

(signature :atomic-concepts (employee )
           :roles ((has_emp :transitive t)
               (has_team :transitive t))
           :individuals(michel sachin ralph sam))

(instance michel employee)
(instance ralph employee)

(instance sachin employee)
(instance sam employee)

(related ralph michel has_emp)
(related michel sachin has_emp)
(related michel sam has_emp)

(related michel sachin has_team)
(related michel sam has_team)
```

In order to test the complex concepts in RACER knowledgebase we extend our knowledge-base with new complex concept called manager, which is an extension of employee concept and has_emp role. Similarly, we extend our knowledge-base with new complex concept called supervisor. The extended knowledge-base with added individuals in RACER A-Box is as follows.

```
(in-knowledge-base employee employee-database)

(signature :atomic-concepts (employee
                                  supervisior manager)
           :roles ((has_emp :transitive t)
               (has_team :transitive t))
           :individuals(michel sachin ralph sam))
```

```
(equivalent manager (and employee (some has_emp employee)))
(equivalent supervisior (and employee (some has_team employee)))

(instance ralph manager)
(related ralph michel has_emp)
(instance michel manager)
(related michel sachin has_emp)

(instance michel supervisior)
(related michel sachin has_team)
(related michel sam has_team)

(instance sachin employee)
(instance sam employee)
```

6.3 Query result comparison and evaluation

In order to evaluate the results of nRQL query against the generated SQL query, we compare the result of nRQL query in RACER ABox and result of the SQL query in MySQL database.

Q6.2.1 Test for primitive concept

nRQL Query : `(retrieve (?x) (?x employee))`

nRQL query result from RACER ABox are:
(((?X RALPH)) ((?X MICHEL)) ((?X SAM)) ((?X SACHIN)))

translated SQL: `Select employee.id from employee`

SQL query result from MYSQL database:
michel, ralph, sachin, sam

Here, the result of nRQL query in RACER ABox and result returned from MySQL database are same.

Q6.2.2 Test for primitive concept with instance

nRQL Query : `(retrieve () (michel employee))`

nRQL query result from RACER ABox are:

T

translated SQL:

```
Select employee.id from employee Where employee.id =
'michel'
```

SQL query result from MYSQL database:

michel

Here, the result of nRQL query in RACER ABox returns 'T'(true) while the result returned from MySQL database returns the instance 'michel'. Even though the result are different. It is similar because in query we test for the instance of 'michel' and query result also return instance 'michel', which menas the query result also returns true.

Q6.2.3 Test for complex concept

nRQL Query : `(retrieve (?x) (?x manager))`

nRQL query result from RACER ABox are:

(((?X MICHEL)) ((?X RALPH)))

translated SQL:

```
Select employee.id from employee where employee.id in   (
Select has_emp.manager from has_emp,employee Where
(has_emp.employee = employee.id) And (has_emp.employee in
(Select employee.id from employee)))
```

SQL query result from MYSQL database:

michel, ralph

Here, the result of nRQL query in RACER ABox and result returned from MySQL database are same.

Q6.2.4 Test for compound 'and' query and primitive role

nRQL Query :

```
(retrieve (?x ?y)
(and (?x manager) (?y employee) (?x ?y has_emp)))
```

nRQL query result from RACER ABox are:

```
(
((?X MICHEL) (?Y SACHIN))
((?X RALPH) (?Y MICHEL))
((?X RALPH) (?Y SACHIN)))
```

translated SQL:

```
Select employee.id from employee where employee.id in   (
Select has_emp.manager from has_emp,employee Where
(has_emp.employee = employee.id) And (has_emp.employee in
(Select employee.id from employee)))
```

SQL query result from MYSQL database:

```
michel, sachin
ralph,michel
ralph, sachin
```

Here, the result of nRQL query in RACER ABox and result returned from MySQL database are same.

Q6.2.4 Test for compound 'or' query

nRQL Query :

```
(retrieve (?x) (or (?x manager) (?x supervisior)))
```

nRQL query result from RACER ABox are:
(((?X MICHEL)) ((?X RALPH)))

translated SQL:

```
Select employee.id from employee where employee.id in   (
Select has_emp.manager from has_emp,employee Where
(has_emp.employee = employee.id) And (has_emp.employee in
(Select employee.id from employee)))
UNION
Select employee.id from employee where employee.id in   (
Select has_team.supervisior from has_team,employee Where
(has_team.employee = employee.id) And (has_team.employee in
(Select employee.id from employee)))
```

SQL query result from MYSQL database:
ralph,michel

Here, the result of nRQL query in RACER ABox and result returned from MySQL database are same.

Q6.2.4 Test for compound 'and' query

nRQL Query :

```
(retrieve (?x) (and (?x manager) (?x supervisior)))
```

nRQL query result from RACER ABox are:
(((?X MICHEL)))
translated SQL:

```
Select employee.id from employee where employee.id in   (
Select has_emp.manager from has_emp,employee Where
(has_emp.employee = employee.id) And (has_emp.employee in
(Select employee.id from employee)))
Intersect
```

```
Select employee.id from employee where employee.id in   (
Select has_team.supervisior from has_team,employee Where
(has_team.employee = employee.id) And (has_team.employee in
(Select employee.id from employee)))
```

SQL query result from MYSQL database:
michel

Here, the result of nRQL query in RACER ABox and result returned from MySQL database are same.

6.4 Evaluation

The above comparison of result for different set of nRQL queries and generated SQL queries shows that our translation are correct and nRQL query to SQL query conversion is valid.

Chapter 7
Conclusion

nRQL to SQL conversion system is able to query individuals from database without maintaining RACER ABox. This enables to query knowledge stored in DBMS, provided that schema for knowledge-base is developed in RACER. This also enables to store large number of individuals of RACER ABox in DBMS, provided that knowledge-base schema are stored in DBMS.

7.1 Future work

7.1.1 Integration with IDE

With integrated IDE user will able to create good knowledgebase. The process of creating knowledgebase from information stored in database can be improved significantly by automatically loading database schemas into RACER knowledgebase.

7.1.2 Flexible storage for RACER ABox

nRQL to SQL conversion system just allows us to query individuals stored in DBMS. Unfortunately it does not support storage of RACER ABox individuals. In real world application, we often need to add, delete and update RACER A-Box individuals. Flexible storage techniques have to be developed, so that it can add, delete and update the RACER ABox individual in the DBMS.

7.1.3 Further nRQL query support

nRQL to SQL conversion system just supports nRQL query for concept query atoms, role query atoms and complex query with ‘and’, ‘or’ and ‘some’. nRQL query can support more these query expression. Further new techniques have to be developed to support other nRQL query expression such as told value query expression and constraint query atoms which could be very useful to query features of tables as well as features of concept in RACER knowledge-base.

7.1.4 Support of other main DBMS

The current implementation of nRQL to SQL conversion system is based on MAXDB MySQL database. It would be, however, very interesting also to try other DBMS such as Oracle, MsSQL and PostgreSQL to compare them with each other in terms of performance efficiency and memory usage.

Bibliography

[1] Sherry Shavor ,*The Java developer's guide to Eclipse,* Addison-Wesley,2003.

[2] *RACER User's Guide and Reference Manual*
http://www.sts.tu-harburg.de/~r.f.moeller/racer/racer-manual-1-7-19.pdf

[3] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 3rd edition, 2003.

[4] Alex. Borgida, Ronald J.Brachaman. Beeri, AT&T Bell Laboratories.Loading Data into Description Reasoner. In *PODS '93: Proceedings 1993 ACM SIGMOD Intern. Conference on Management of Data Washington, DC., USA, May 1993.*

[5] R. Ramakrishnan O. Shmueli C. Beeri, S. Naqvi and S. Tsur. Sets and negation in a logic data base language (ldl1). In *PODS '87: Proceedings of the sixth ACM SIGACTSIGMOD-SIGART symposium on Principles of database systems*, pages 21–37, New York, NY, USA, 1987. ACM Press.

[6] Paolo Bresciani. The Challenge of Integrating Knowledge Representation and Database. In *PODS '96: International journal of Computing and Informatics*, pages 443–453, Informatica (ISSN 0350-5596).

[7] Jeffrey D. Ullman. Information Integration Using logical Views*.

[8] H. Gallaire and J. Minker. *Logic and Databases*. Plenum Press, New York, 1978.

[9] Ullman, J.D. Principles of Database and Knowledge-Base Systems,Vol.I. Computer Science Press, New York.

[10] Ullman, J.D. Principles of Database and Knowledge-Base Systems,Vol.II. Computer Science Press, New York.

[11] K. Je¡zek and V. Toncar. Deductive Database Implementation using RDBMS. In *AIT'96Workshop*, Brno, 1996.

[12] Kelvin Kline and Daniel Kline. *SQL In a Nutshell*. O'REILLY publication, 1rd edition, 2001.

[13] Andrea Cali, Diego Calvanese, Giuseppe De Giacomo and Maurizio Lenzerini. Accessing Data Integration System through Conceptual Schemas*(extended abstraction). In *PODS '2001: Proceedings of the 20th Int.Conf on Conceptual modeling*.

[14] Enrico Franconi. *Logicsfor Information modeling and access*, Lecture notes from Faculty of Computer Science, Free University of Bozen-Bolzano.

Software tools

• RACER 1.8.x Server (Windows) **, V. Haarslev, R. Möller, M. Wessel (*http://www.sts.tu-harburg.de/~r.f.moeller/racer/racer-1-7-23-windows).*

RICE , RACER Interactive Client Environment, Academic Medical Center, dept. of Medical Informatics (*http://www.b1g-systems.com/ronald/rice)*

J2SE 1.4, Java application development platform, Sun Microsystems *(http://dlc.sun.com/jdk/j2sdk-1_4_2_07-windows-i586-p.exe)*

Eclipse 3.0, open extensible IDE, Eclipse Foundation *(http://www.eclipse.org/downloads/index.php)*

MySQL 5.0, open source Database server, MySQL AB *(http://www.mysql.com/products/database/)*

Appendix

A. “Family.racer” knowledge base file (TBOX & ABOX)

```
(in-knowledge-base family smith-family)

(signature :atomic-concepts (human person female male woman man
                                   parent mother father
                                   grandmother aunt uncle
                                   sister brother)
           :roles ((has-descendant :transitive t)
                   (has-child :parent has-descendant)
                   has-sibling
                   (has-sister :parent has-sibling)
                   (has-brother :parent has-sibling)
                   (has-gender :feature t))
           :individuals (alice betty charles doris eve))

(implies *top* (all has-child person))
(implies (some has-child *top*) parent)

(implies (some has-sibling *top*) (or sister brother))
(implies *top* (all has-sibling (or sister brother)))
(implies *top* (all has-sister (some has-gender female)))
(implies *top* (all has-brother (some has-gender male)))

(implies person (and human (some has-gender (or female male))))
(implies woman (and person (some has-gender female)))
(implies man (and person (some has-gender male)))

(equivalent parent (and person (some has-child person)))
(equivalent mother (and woman parent))
(equivalent father (and man parent))

(equivalent grandmother
          (and mother
             (some has-child
                   (some has-child person))))
(equivalent aunt (and woman (some has-sibling parent)))
(equivalent uncle (and man (some has-sibling parent)))

(equivalent brother (and man (some has-sibling person)))
(equivalent sister (and woman (some has-sibling person)))

(instance alice mother)
(related alice betty has-child)
(related alice charles has-child)

(instance betty mother)
(related betty doris has-child)
(related betty eve has-child)
```

```
(instance charles brother)
(related charles betty has-sibling)
(instance charles (at-most 1 has-sibling))

(related doris eve has-sister)

(related eve doris has-sister)
#|
(concept-subsumes? brother uncle)

(concept-ancestors mother)

(concept-descendants man)

(all-transitive-roles)

(individual-instance? doris woman)

(individual-types eve)

(individual-fillers alice has-descendant)

(individual-direct-types eve)

(concept-instances sister)

|#
```